D2.3: Voice Programming of Devices and Services

Oliver Lemon, Kallirroi Georgila, David Milward, Tommy Herbert

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable 2.3

January 11, 2007

Project funded by the European Community
under the Sixth Framework Programme for
Research and Technological Development

The deliverable identification sheet is to be found on the reverse of this page.

Project ref. no.	IST-507802
Project acronym	TALK
Project full title	Talk and Look: Tools for Ambient Linguistic Knowledge
Instrument	STREP
Thematic Priority	Information Society Technologies
Start date / duration	01 January 2004 / 36 Months

Security	Public
Contractual date of delivery	M36 = December 2006
Actual date of delivery	January 11, 2007
Deliverable number	2.3
Deliverable title	D2.3: Voice Programming of Devices and Services
Type	Report
Status & version	Final 1.0
Number of pages	38 (excluding front matter)
Contributing WP	6
WP/Task responsible	UEDIN
Other contributors	LING
Author(s)	Oliver Lemon, Kallirroi Georgila, David Milward, Tommy Herbert
EC Project Officer	Evangelia Markidou
Keywords	voice programming, devices, services, in-car, in-home

The partners in TALK are:

Saarland University	USAAR
University of Edinburgh HCRC	UEDIN
University of Gothenburg	UGOT
University of Cambridge	UCAM
University of Seville	USE
Deutches Forschungszentrum fur Künstliche Intelligenz	DFKI
Linguamatics	LING
BMW Forschung und Technik GmbH	BMW
Robert Bosch GmbH	BOSCH

For copies of reports, updates on project activities and other TALK-related information, contact:

The TALK Project Co-ordinator
Prof. Manfred Pinkal
Computerlinguistik
Fachrichtung 4.7 Allgemeine Linguistik
Postfach 15 11 50
66041 Saarbrücken, Germany
pinkal@coli.uni-sb.de
Phone +49 (681) 302-4343 - Fax +49 (681) 302-4351

Copies of reports and other material can also be accessed via the project's administration homepage, `http://www.talk-project.org`

Contents

Summary

This deliverable reports work in the task, “Programming Devices and Services”, which is focussed on allowing users to reconfigure dialogue systems by using spoken dialogue to build simple programs for the devices and services they use. This is a particular type of adaptivity – where the user is able to explicitly adapt some aspects of the dialogue system to their needs. The idea is to extend command-based dialogue interfaces and information-seeking dialogue systems so that users can reconfigure them to perform common tasks, or to behave in specific ways in certain contexts that are of interest to the user.

This is a different type of adaptivity from Task 2.2 (Dynamic Reconfiguration) [MAB+06] which focusses on “plug-and-play” style dialogue capabilities of devices and services.

We present 3 voice-programming (VP) scenarios:

- VP for information seeking services - the extended TownInfo system
- VP for in-car devices
- VP for in-home devices

The first 2 of these scenarios have a demonstration system which is built on the baseline system presented in D4.2 [LGS05]. This demonstration system uses software tools developed in other parts of the TALK project, for example we used GF (UGOT, WP1), ATK (UCAM, WP1), the MySql database agent (DFKI, WP5), and the display agent (DFKI, WP5), from the TALK project software infrastructure [BPL+06]. The prototype system allows voice programming of macros, conditionals, and event-driven programs. As far as we are aware, no commercial or research system has before implemented these functionalities, and we survey the related work of [LL05b, LL05a, LKB+02, TGS06]. The Voice Programming described in this deliverable was presented and demonstrated at Brandial 2006 [GL06].

The third scenario, Voice Programming for in-home devices explores a different approach where user defined macros are directly added to the system ontology. This was provided as an extension to the Linguamatics Interaction Manager.

Both approaches use the advantages of ISU-based dialogue systems, where the modularity of Information States and Update Rules has allowed us to develop domain-independent Voice Programming methods.

Chapter 1

Introduction

Task 2.3 of the TALK project, "Programming Devices and Services", is focussed on allowing users to reconfigure dialogue systems by using dialogue to build simple programs for the devices and services they use. The idea is to extend command-based dialogue interfaces and information-seeking dialogue systems so that users can explicitly reconfigure them to perform common tasks, or to behave in specific ways in certain contexts that are of interest to the user. This is a particular type of adaptivity – where the user is able to use spoken dialogue to adapt some aspects of the dialogue system to their needs.

The deliverable is structured as follows:

- Chapter 2 is a presentation and discussion of Natural Language programming constructs that could be useful in natural interactions with devices and services, for example conditionals, loops, event-driven programs, and macros. This chapter also provides and analyses some example in-car and in-home multimodal dialogues showing these constructs.
- Chapter 3 presents the Wizard of Oz data collection that we performed for exploring voice-programming (VP) tasks, and the observations we made from the data.
- Chapter 4 discusses the technical challenges to be overcome in implementing these new types of dialogue in a generic way, using ISU-based dialogue systems (for example new data structures in Information States, new update rules, and new grammar rules).
- Chapter 5 presents the extended TownInfo voice programming system, for macros, conditionals, and event-driven programs over services and in-car devices, and its development details.
- Chapter 6 discusses how voice programming can be implemented using an ontology based approach and presents examples of voice programming using the Linguamatics Interaction Manager with a home information and control application.
- Chapter 7 presents our conclusions and challenges for future work.

It is one of the advantages of the ISU approach to dialogue adopted in the TALK project that we can develop a domain-general approach to voice programming which implements voice programming without any recourse to specific knowledge about the application domain.

In the Edinburgh "TownInfo" system the addition of Voice Programming utilizes new structures in the Information States, and new Update Rules. Application-specific knowledge is localized to the grammar

rules and language model for specific applications, as we show in chapters 4 and 5. In the Linguamatics system the addition of Voice Programming utilizes the ability to dynamically create new structures within the ontology.

We now conclude the introduction with a comparison of our approach with current and recent related research [LKB+02, LL05b, LL05a, TGS06].

1.1 Comparison with "state-of-the-art" / Related work

We are not aware of any prior commercial or research system which allows the construction and execution of "voice programs". However, there are several strands of research that deal with different aspects of programming using natural language, and we survey these below.

1.1.1 The Metafor project

In the Metafor project [LL05b, LL05a] Liu and Lieberman explore the idea of using descriptions typed in natural language as a representation for programs (real programming code). See figure 1.1 for an example of the interface.

Metafor cannot yet convert arbitrary English to fully specified code, but it uses a reasonably expressive subset of English as a visualization tool. Simple descriptions of program objects and their behaviour generate scaffolding (underspecified) code fragments, that can be used as feedback for the designer. The system does not manage to transform natural language to fully-formed code but rather serves as a tool for designing a program or a brainstorming tool.

1.1.2 Instruction based learning

"Instruction based learning" (IBL) for mobile robotics [LKB+02] also deals with some of the same issues. Here a robot is given route descriptions such as "turn left after the church" which in some sense are programs for navigation in particular environments. This is most similar to "event-driven" voice programs, see chapter 2. However, the IBL system does not support the user in defining their own named macros, conditionals, or event driven programs that react to events other than the robot's own progress in navigation/localization.

1.1.3 Programming by demonstration

In the "GSI Demo" system [TGS06] users can associate speech commands with keyboard and mouse actions, to support tangible interactions with Digital Tables. An example provided by the authors is "Computer, when I say [speech command], you do [keyboard and mouse sequence]". Users train the system by demonstrating the speech actions it should recognise and then performing the appropriate mouse and keyboard actions.

The GSI system is restricted in various ways. First, each user must have a saved speech profile (the system uses Microsoft SAPI). Second, speech commands are actually *typed* to the system when training it, rather than being spoken. The defined mappings are then stored in a list for later retrieval and may also be saved for later use, by using a GUI.

Figure 1.1: The Metafor system interface

1.2 Discussion

Considering the above survey of current and recent related work, we can see that the area of "programming in natural language" is only just beginning to emerge, and that the functionality which we propose, and develop below, is novel. There is currently a small number of systems that deal with different parts of the issue in different ways.

Our approach (as presented in this report) is perhaps most similar to that of GSI Demo and IBL, in the sense that for in-car and in-home dialogue systems we wish the user to be able to use natural language (in our case speech input) to teach the system instructions about how to behave (i.e. to perform a sequence of actions) based on user-defined commands or other trigger events.

In the next chapter we will survey the "Natural Language programming" constructs that could be useful in interactions with both devices and services in the car and in the automated home.

Following this conceptual analysis, we show data from a Wizard-of-Oz (chapter 3) experiment, which has driven the development of TALK's prototype Voice Programming system of chapter 5.

Chapter 2

Programming in Natural Language

Here we survey Natural Language (NL) programming constructs that could be useful in natural interactions with devices and services, for example conditionals, loops, event-driven programs, and macros.

2.1 Programming devices and services

The majority of users do not want to deal with learning complex operating instructions for their electronic devices. They want to be able to activate and control a number of devices by just giving simple commands (in speech, via graphics, or by a combination of modalities). Moreover, users may also wish to configure their devices to carry out specific actions which are tailored to their own needs and preferences. One way of doing this is to allow users some level of "programming" access to the interfaces themselves.

For example, in a home-automated environment, by uttering a simple command such as "activate relax mode" a user may request the home control system to perform a number of pre-defined tasks such as turn down the lights, play classical music, and switch the telephone device to the automatic answering machine mode. In a similar way, in the car, when a sensor detects rain, the driver may have configured the control system to respond to such events by automatically closing the windows and activating the windscreen wipers.

The idea of programming services is similar to the idea of programming devices described above. It is much faster, easier, and more robust for the user to say, for example "show me my favourite" than "show me all expensive French restaurants in the centre of the city". This is not only a matter of convenience for the user but also an efficiency issue. Considering speech recognition limitations especially in noisy environments such as cars, shorter and more precise commands will in general lead to fewer errors and increase overall user satisfaction.

In this research we will focus on controlling devices and services by using programs which are:

- activated by speech commands or environmental events
- defined by the user through use of speech dialogues.

A program prescribes the actions that are to be carried out by the system when the user gives a command or if a particular event takes place (event-driven programming). These actions may be completely independent of each other or be related with logical operators such as "and" and "or", be parts of "if-then clauses", "for" and "do-while" loops etc.

In the following we divide programs for controlling devices and services into 5 categories inspired by similar classifications used in computer programming literature. In each category commands for controlling devices and services can be combined. The terms 'action' and 'command' are used in a standard way. Thus "turn off the lights" is considered as a command and the system performs an action when it actually turns off the lights.

2.1.1 Macros

A macro is a way for the user to automate a complex task that he/she performs repeatedly or on a regular basis. It is a series of commands that can be stored and run whenever the user needs to perform the task. The user can record or build a macro, and then play the macro to automatically activate the series of actions.

The syntax for a macro is:

```
macro_name = command_1 ( and ) command_2 (and ) ... command_N
```

In our case an example of a macro definition at home would be: "When I say 'movie mode' please turn off the lights, turn on the DVD player and the TV, close the curtains, and turn off the stereo".

Similarly in the in-car domain a macro could be: "When I say 'romantic dinner' I mean dinner for two people at an expensive Italian restaurant with music" or "When I say 'wake me up' I mean turn the music to full volume and turn on cold air".

An alternative way of defining macros is to use "macro recorders". These provide users with a way to record their actions. These recorders are a basic implementation of "Watch what I do". The user issues the "Start Recording" command, performs a series of actions, and then issues the "Stop Recording" command. All of the user's actions are saved as a sequence, then the user can give a name to this sequence that will be saved as a macro. Many spreadsheet programs and telecommunications programs have built-in macro recorders [Cyp93]. In this project, however, we focus on explicit definition of macros as shown in the examples above.

Our proposal is then to extend the coverage of our system to interpret some types of user utterances as macro definitions.

For example user inputs such as:

- When/If/Whenever I say C, then X_1 ... X_n
- X_1 ... X_n when/if/whenever I say C

will be interpreted as defining a macro with trigger phrase C and which stands for commands X_1 ... X_n.

Note however that in terms of the dialogue context, the effect of X_1 ... X_n is not the same as if the user had actually uttered these commands. For example the salient NPs in X_1 ... X_n should not be available for anaphoric reference. Exactly what the effects on the dialogue context should be is a matter for ongoing research.

Note also that some macro definitions are more naturally understood as requests that certain *states* obtain rather than actions be performed. Here we may require inference in context to determine the actual sequence of actions required to achieve the desired state. We leave this issue for future work in voice programming research.

2.1.2 Loops and iteration

In standard programming, loops are used when an action/command or a series of actions/commands must be repeated more than once. Generally there are two types of loops, “for” and “do-while” loops.
The syntax for a “for” loop is:

```
for i=1 to N {
   command_a
   command_b
   ...
   command_z
}
```

Which means that the sequence of (command_a, command_b, ..., command_z) will be carried out N times.
The syntax for a “do-while” loop is:

```
do {
   command_a
   command_b
   ...
   command_z
} while (loop_condition=true)
```

A variation of the “do-while” loop is the “while” loop:

```
while (loop_condition=true) {
   command_a
   command_b
   ...
   command_z
}
```

The difference is that in the first case the program will perform the series of commands at least once before it checks the validity of the loop_condition.

2.1.3 Conditionals

Conditionals are “if-then” clauses. Thus the syntax of a conditional is as follows:

```
if (condition=true) then {
   command_1
   command_2
   ...
   command_N
}
```

or

```
if (condition=true) then
   execute macro_name
```

Examples of conditionals for programming devices and services could be:

- "When I ask for pizza always make it expensive."
- "If I say 'relax mode' then play my favourite CD."

This last example illustrates how conditionals and macros can be combined. The system could expand the "relax mode" conditional taking into account the definition of the macro "favourite CD".

2.1.4 Event driven programming

Traditionally, programs operate in a sequential fashion: they do some processing, produce output, perhaps wait for a user response, do some more processing, and so on. Event-driven programming is a flexible way to allow programs to respond to many different inputs or events. Instead of tightly controlling the run of a program, the software waits until the user does something or an event takes place. In other words, the flow of the program is controlled by user-generated or external events. In our case, an external event generated independently of the user's behaviour could be an environmental event such as a sensor detecting rain or freezing temperatures, or the driver behaving erratically.

The need for event-driven programs arose from the introduction of GUI environments. The core idea of event-driven programming is in the event-handlers. These are functions in the software that are called in response to certain events. Thus the concept of event-driven programming is that the user or some external action is in control. That means that the system cannot be expected to follow a predefined execution path. The system must be open-ended and more robust error handling is required. It cannot be assumed that the user has entered data in a certain order or produced one event before another, etc.

From now on, to make things clear, we will refer to event-driven programs only when the event is external and independent of the user's behaviour. If the event is a user command it will be considered as a conditional.

2.1.5 Programming by example

Programming by example is an elaboration of the idea behind macro recorders given in section 2.1.1. Once again, the user instructs the system to "Watch what I do", but with programming by example, the system creates generalised programs from the recorded actions [Cyp93].

For example suppose the user selects all U2 live songs from a list of live recordings and then defines this sequence of actions as a macro called 'favourite live music'. If new U2 live songs were added to the list a macro would fail to include the new songs in its list of actions. However, a program based on the idea of "programming by example" would generalise this and associate 'favourite live music' with all U2 live songs and not just the ones that were selected by the user during the definition of the program.

2.2 Example dialogue scenarios

This section provides and analyses some example in-car and in-home multimodal dialogues showing the natural language constructs described above. We presented some examples while introducing our 5 program categories. However, in this section we will describe more complex examples that illustrate how different program categories could be combined to form advanced flexible programs for controlling devices and services.

2.2.1 Programming devices

In car

Here we survey some possible voice-programming examples in the in-car scenario:

> Example 1:
> "When I say 'quiet mode' close the windows, turn on the air-conditioning and turn off the radio."

In the above example the user defines the macro "quiet mode". This macro is actually a combination of three commands connected ('close the windows' , 'turn on the air-conditioning' and 'turn off the radio').

> Example 2:
> "If I say 'turn off the radio' always turn off the DVD player as well."

This is a conditional. It is driven by a user command and not by an external event, which is why we classify it as a conditional (see section 2.1.4).
Note the difference between a macro and a conditional. A macro has similar syntax to a conditional (if I say 'quiet mode' then do a list of actions). However, the condition (if-part) of a macro is an arbitrary name and not a user command with meaning ('turn off the radio'). The then-part of the macro gives meaning to its arbitrary name.

> Example 3:
> "When it is raining activate the windscreen wipers and close all windows."

This is a program driven by the event of rain.

In home

Here we survey some possible voice-programming examples in the smart home scenario:

> Example 1:
> "Every time I ask you to activate the alarm system make sure to close windows and turn off all lights."

This is a conditional combined with two loops. One loop closes all windows and another loop turns off all lights.

> Example 2:
> "When I say 'safe mode' activate the alarm system."

In the previous example the user defines the macro 'safe mode'. Now if in the course of the dialogue the user says 'safe mode', the alarm system will be activated. This will trigger the conditional of example 1 and all windows will be closed and all lights will be turned off.

2.2.2 Programming services

Here we survey some possible voice-programming examples for services. The "TownInfo" service can be used in both in-car and in-home scenarios:

Example 1:
"When I say 'special hotel' search for a double room in a luxurious central hotel."

Example 2:
"When I say 'program three' display all Irish pubs in the centre and find me a special hotel."

The latter example is the definition of a macro as a combination of a loop and another already defined macro.

Example 3: "If I order pizza make it expensive."

This is a conditional. Every time the user wants to order pizza he/she expects the system to remember that it should be from an expensive restaurant. This example raises a series of questions. What if the user explicitly says that he/she wants cheap pizza? We believe that the system should ignore predefined conditionals if they are in conflict with current requests. Another question that may arise is what will happen if the user orders pizza but does not remember that he had previously defined a conditional associated with the term 'pizza'. It is impossible in this case for the system to know in advance whether the user means 'expensive pizza' or just 'pizza'. In order to cover the case that the user has forgotten about his/her predefined programs, we believe that the system should *implicitly confirm* that it will order 'expensive pizza' and thus give the user the opportunity to accept the order as it is or modify it. Thus, in our system, described in chapter 5, activated voice programs are always confirmed to the user when they are being activated.

2.3 Summary

In this chapter we have surveyed typical programming constructs that could be useful in natural language interactions with devices and services, for example conditionals, loops, event-driven programs, and macros. We then presented some more complex examples that illustrate how different program categories can be combined to form advanced flexible programs for controlling devices and services in the car and in the home.

We now present a Wizard-of-Oz study that moves us from this conceptual analysis to data from real human "Voice Programming" behaviour, which drives our later system development (see chapter 5).

Chapter 3

A Wizard of Oz data collection for Voice Programming

In this chapter we describe a small “Wizard of Oz” (WoZ) experiment conducted to discover the kinds of natural language constructs that users might produce when attempting different types of VP tasks. This study was conducted in the smart-home and tourist-information domains, where users are interacting both with devices in an automated house, and an automated tourist information service (as in the TALK in-car systems of [LGS05, YST+06, LGHS06]).

In the Wizard-of-Oz setup, a human (in a different room) played the part of the voice programming (VP) dialogue system, using the Festival system to synthesise speech, so that the human participant was unaware that they were in fact speaking to another human. The wizard also controlled the state of in-home devices on the subjects’ GUIs (see section 3.2).

3.1 Tasks

For the in-home domain, subjects were presented with the following VP tasks:

1. You’re worried about your bills, and you want the heating to be linked to the outside air temperature. Arrange things with the computer so that the heating turns off whenever the temperature rises above 15 degrees.

2. You want to make an appointment at the hospital. The trouble is, so does everyone else, so you keep getting an engaged tone when you call the number. Get the computer to keep trying until they answer, so you can go and do something else while you wait.

3. You’ve bought a new widescreen TV and a DVD player for the bedroom. Let the computer know this, so it can add them to its list of voicecontrolled appliances.

4. When you want to test whether something works, you usually turn it on for a couple of seconds to check, and then turn it off again. Teach this procedure to the computer, so that you can just use shortcut commands like ”test the radio” and ”test the TV”.

5. It would be nice to be able to give a single command phrase, say ”movie mode”, to turn on the heating, the TV and the sitting room lights and turn everything else off. Teach this phrase to the computer.

For the tourist information service, subjects were presented with the following VP tasks:

1. Whenever you want a romantic meal, you tend to ask for an expensive Italian restaurant in the town centre. Tell the computer that that’s what ”romantic meal” means.
2. Arrange things so that whenever you ask for a Chinese restaurant, the computer always recommends somewhere cheap.

10 participants were recruited, and they did each of these tasks twice, resulting in 140 dialogues.
For the in-home tasks, subjects could monitor the behaviour of their (simulated) smart home, using a GUI described in the next section.

3.2 The In-home simulation GUI

The in-home simulation GUI (figure 3.1) allowed subjects to monitor the state of their (simulated) smart home, and see the results of their voice programs. It allowed subjects to see the state of lights, appliances, and heating in the different rooms of the house.
This was implemented as an OAA agent, using the Display Agent from the TALK software infrastructure [BPL+06].

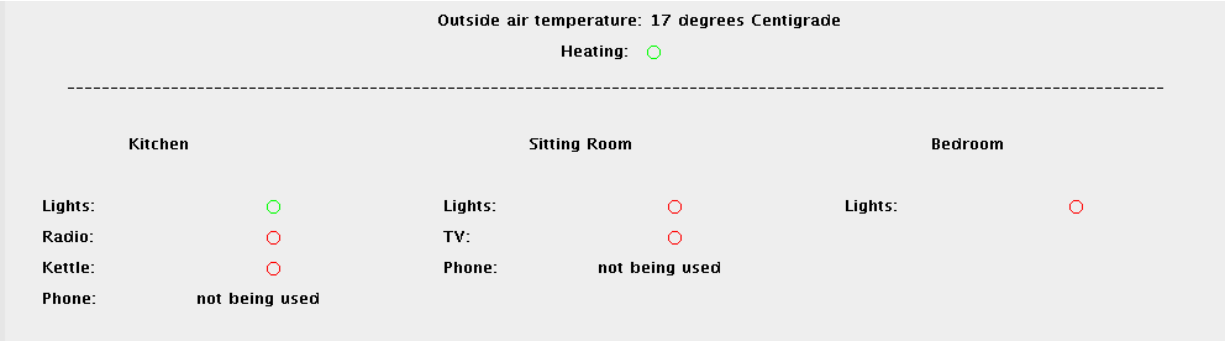

Figure 3.1: The “in-home” WoZ collection system GUI

3.3 Results

In general, the WoZ exercise revealed a surprising range of possibilities for expressing voice programs. For example, we observed eight ways of saying “if X then Y”:

- “When(ever) X, Y.”

- “I would like you to Y when X.”
- “If X, Y.”
- “Y when(ever) X.”
- “You should Y only if X.”
- “I’d like you to Y every time X.”

Here are some other frequently observed user inputs to the WoZ system, which we used for developing the GF grammar for Voice Programming (see chapter 4):

- movie mode means turn on the TV in the sitting room
- when I say movie mode, turn on the TV in the sitting room
- when I say movie mode, I mean turn on the TV in the sitting room
- when I say movie mode, it means turn on...
- when I say movie mode, I want you to turn on...
- when I say movie mode, please turn on...
- when I say movie mode, you have to turn on...
- when I ask/tell you to be in movie mode, I want you to turn on...

3.4 Summary

For system development, we wanted to discover the types of utterances users would use in spontaneous dialogues for programming both devices and services. In order to collect relevant data for this purpose we developed a “Wizard-of-Oz” setup and ran 10 subjects in a variety of voice-programming scenarios.

We then analysed the various linguistic forms used by the subjects, to discover the range of inputs and dialogue structures that our Voice Programming systems should aim to cover.

We now move on to discuss the technical challenges involved in implementing these types of Voice Programming in a general way, using ISU-based dialogue systems.

Chapter 4

Extending Information States for Voice Programming

This chapter discusses the technical challenges to be overcome in implementing VP dialogues in a general way by creating new data structures in Information States and new Update Rules for maintaining the dialogue context. In addition, we describe how language models and grammars are extended to allow for VP. In chapter 5 we also describe our approaches to handling these challenges in an extended version of the "in-car" dialogue system [LGS05, LGHS06].

4.1 Extended Information States

New fields must be added to Information States (IS) to handle "voice programming" but the interesting challenge is to do this in a domain-independent way.

The central idea is that the Update Rules (in DIPPER [BKLO03]) that handle the additional IS fields should abstract over names of specific devices and macros etc. and therefore will not depend on any specific domain. To do this we define Macros, Conditionals, and Event-driven programs using stacks of lists in the IS, where macro and device names etc. appear as variables.

For example, for programming services the macro definition "when I say my 'favourite' I mean expensive Italian restaurant" can be stored in the Information State as follows:

```
macros: < ('([favourite],u)',restaurant,['([food_type],s)','([price_range],s)'],
                                                    [[italian],[expensive]]) >
```

where the first argument denotes the name of the macro, the second argument shows the task, and the third argument is a list containing the definition of the macro.

In the same way, for devices the macro definition "if I say 'quiet mode' it means turn off the radio and close the windows" can be stored in the information state as follows:

```
macros: < ('([quiet_mode],u)',device,['([turn_off_radio],s)','([close_windows],s)'],
                                                    [[off],[close]]) >
```

The same idea is applied to conditionals. For services, the program “when I ask for pizza always make it expensive” will be stored in the Information State as follows:

```
conditionals: < (restaurant,['([food_type],s)'],[[pizza]],
                                ['([price_range],s)'],[[expensive]]) >
```

Also, for devices, the program “if I ask you to open the roof always turn the radio off” will be stored in the information state as follows:

```
conditionals: < (device,['([roof],s)'],[[open]],
                                ['([radio],s)'],[[off]]) >
```

As explained in section 2.1.4 conditionals are much like event-driven programs. In the above conditional we assume that the user asks the system to open the roof. It could also be the case that the roof was connected to a sensor, so that every time it opened (e.g. by pressing a button), the system would turn off the radio. Thus event-driven programs can be defined in the same way as conditionals and become activated with small modifications in the dialogue system to support interrupts or polling.

The current system implementation can support polling. When in simulation mode it can randomly change the status of an event in the beginning of the dialogue, e.g. assume that it is very windy. In the beginning of the dialogue, the system will loop (polling) through all supported events, checking if an event has been detected. Then it will search through the stored event-driven programs to see if an event that has been detected can trigger other events/actions. In the following example of the ’windy’ event, the events/actions ’close the windows’ and ’close the roof’ will also be triggered:

```
event-driven programs: < (event,['([windy],s)'],[[yes]],
                     ['([windows],s)','([roof],s)'],[[close],[close]]) >
```

4.2 New Update Rules

We also need new Update Rules to support the development of voice programs. In addition, once these programs have been defined we need Update Rules that can retrieve the programs from the information state and activate them once the parser has detected that the user input is associated with a macro, conditional, or event-driven program. Furthermore, once e.g. a macro has been detected the slots that are associated with this macro must be automatically filled and confirmed. In the following we can see an example Prolog predicate definition from part of an Update Rule, which is a generic operation that maps a macro definition (stored in the IS as shown above) to the slots that must be filled:

```
mapmacros2filledslots(CurrTask,[],[],FilledSlotsNew,FilledSlotsValuesNew,
                                          FilledSlotsNew,FilledSlotsValuesNew):-!.


mapmacros2filledslots(CurrTask,[Hfs|Tfs],[Hfsv|Tfsv],FilledSlots,FilledSlotsValues,
                     FilledSlotsNew,FilledSlotsValuesNew):-
      macros_def(Hfs,Task,SlotsList,ValuesList),
      Task==CurrTask,!,
```

```
      append(FilledSlots,SlotsList,FilledSlots1),
      append(FilledSlotsValues,ValuesList,FilledSlotsValues1),
      mapmacros2filledslots(CurrTask,Tfs,Tfsv,FilledSlots1,FilledSlotsValues1,
                                              FilledSlotsNew,FilledSlotsValuesNew).


mapmacros2filledslots(CurrTask,[Hfs|Tfs],[Hfsv|Tfsv],FilledSlots,FilledSlotsValues,
                        FilledSlotsNew,FilledSlotsValuesNew):-  !,
      append(FilledSlots,[Hfs],FilledSlots1),
      append(FilledSlotsValues,[Hfsv],FilledSlotsValues1),
      mapmacros2filledslots(CurrTask,Tfs,Tfsv,FilledSlots1,FilledSlotsValues1,
                                                FilledSlotsNew,FilledSlotsValuesNew).
```

Again, note that this Update Rule abstracts over macro names, task names, and the particular domain specific details of slots.

Update rules are also required for controlling devices, e.g. the update rule that decides which devices should be turned on or off according to a user's request or an event detection is as follows:

```
urule(turnonoffdevices,
     [
      is^flgcommand=1,
      is^flgchangedevice=0,
      is^voiceflg=0,
      is^flgturn=system,
      prolog(check_slot_devices(top(is^groundedslots),ResSlot)),
      ResSlot=1,
      is^flgsave=0

     ],
     [
      prolog(check_devices(top(is^groundedslots),ChangeDeviceList)),

      push(is^changedevicelist,ChangeDeviceList),

      assign(is^flgchangedevice,1)
     ]
    ).
```

4.3 Extended Language Models

One of the main issues in "programming by voice" is the vocabulary the user may use to form e.g. macros. If the speaker is free to name a macro with any word or phrase he/she can think of, it is possible that this word or phrase will not be included in the dictionary and the language model or grammar used by the speech recogniser. In a very advanced "voice by programming" system the speech recogniser should be able to handle out-of-vocabulary words [Pur02].

However, current speech recognisers fail if they attempt to recognise an utterance that contains out-of-vocabulary or out-of-grammar phrases. Unfortunately they will not be able to detect the problem but instead produce as output the path of their language model or grammar with the highest probability for the given speech input. Only very sophisticated speech recognition systems will be able to spot the speech segment with the out-of-vocabulary word or phrase. We have made initial progress on this with fragmentary clarifications based on word confidence scores form ATK [LGS05, LGHS06]. This means that, in principle, we can localise poorly recognised *parts* of utterances.

However, this is not enough. The ideal system should have a method to add new phrases in the vocabulary and the grammar or the language model. One way could be by asking the user to spell the phrase. Then the system should be able to add it to its vocabulary using a grapheme-to-phoneme converter and to its grammar or language model, which involves additional problems. Where in the grammar should this phrase be added? If we add it to the language model how will the system compute language modelling statistics on the fly?

It is not in scope of this project task to handle such problems. Therefore, in the following we restrict the user to give specific names to macros, e.g. 'favourite restaurant', 'favourite hotel', 'usual bar', 'program one', etc. These names are included in a list of words which we explicitly include in the vocabulary of the system. Currently, new macro names can be added but this can be done only offline by the designer of the application.

4.4 Parsing NL "voice programs"

Defining programs requires more sophisticated parsing than executing these programs especially for complex definitions that include loops, and-or operators, etc. As we explained in deliverable D4.2 [LGS05] we use a keyword parser for our baseline system. This parser has now been extended in order to be able to deal with complex voice programs. The same applies to the language model used for speech recognition. A grammar-based language model is expected to be much more accurate than a statistical based one for the task of defining programs. This is because the grammar-based model will always produce hypotheses that can be easily mapped to programs whereas a statistical language model could produce results that are more difficult to parse.

4.5 Extending the recognition grammar

In order for the dialogue system to recognise various types of voice programming utterances, it is necessary to develop either a statistical language model or a grammar-based language model for speech recognition. Since we did not have enough "Voice Programming" data to develop a good statistical model, we decided to use the grammar-based language modelling capabilities of the Grammatical Framework system [Ran04] as developed and explored in TALK workpackage 1.

We developed a GF grammar for normal information seeking utterances such as "I am looking for a luxury hotel in the centre of town" and to this grammar we then added lexical items for voice programming (i.e. macro and trigger phrases), and specific rules to cover linguistic constructs for voice programming. These linguistic constructs were derived from our WoZ experiments as discussed in chapter 3.

An example of a GF rule for voice programming is the following:

```
UttRulePhrs79   :
     VPStarter -> VPTrigger -> VPMacro ->  Article -> Utt -> Utt -> Utt  -> Phrs
         =  \vpsttr,vptrig,vpmacro,artcl,uttstr1,uttstr2,uttstr3 ->
            {s = vpsttr.s++vptrig.s ++ vpmacro.s ++  artcl.s ++ uttstr1.s
                   ++ uttstr2.s ++ uttstr3.s ; point = []};
```

This allows us to recognise and parse user utterances such as “When I say (a VPStarter) romantic night (a VP Trigger) it means (a VP Macro) a (Article) luxury French restaurant in the town centre (a sequence of 3 Utts)”.

Items such as VPTriggers (e.g. “program one”, “quiet mode”, “program two”...) are listed in the grammar’s lexicon.

4.6 Summary

This chapter discussed the technical challenges to be overcome in implementing Voice Programming in a generic way in ISU dialogue systems (for example new data structures in Information States and new Update Rules for maintaining the dialogue context). We also explained the use of GF for building a grammar-based language model for Voice Programming based on our WoZ experiments described in chapter 3. These ideas were implemented in the system described in chapter 5.

Chapter 5

Voice programming “in-car” system for devices and services

This chapter presents an extension of the baseline dialogue system presented in deliverable D4.2 [LGS05] and [LGHS06], based on the WoZ data collection of chapter 3 and the technical ideas presented in chapter 4.
The prototype system now combines:

- VP for information seeking (as reported in [GL06])
- VP for in-car devices.

5.1 Voice Programming for information seeking services (extended TownInfo system)

The system extends that presented in deliverable 4.2 [LGS05] and supports the definition and execution of:

- macros
- conditionals
- event-driven programs.

The parser has been extended, as discussed, to support simple commands such as:

- “When I say ... I mean ...“
- “... means ...”
- “If I say ... make it ...“
- “When I say ... I want you to ...”

- "Every time I tell you to ... you should ..."

The above linguistic structures have been derived from our WoZ experiments described in chapter 3. They are the most common type of sentences used by users to define voice programs. Other structures could be very easily added to the GF grammar, parser, and speech recognizer.

5.1.1 Dialogue management

The baseline (rule-based) version of the system [LGS05] started the dialogue with a general question "How may I help you" and according to the user's reply the dialogue could follow one of 3 branches for the 3 tasks of hotels, bars, and restaurants, respectively. Now a fourth possible dialogue branch has been added for the definition of macros, conditionals and event-driven programs.

Once the user defines a program it is stored in the information state as explained in section 4.1. Now when the user enters the normal mode of information-seeking about hotels, bars, and restaurants, the dialogue manager DIPPER [BKLO03] reads all information stored in the macros and conditionals fields and transforms them into Prolog clauses that are asserted in memory. Moreover, word patterns that will be used by the parser are also formed and loaded. Then, every time the parser processes a user input it searches these new clauses and not only takes into account the standard word patterns but also the user-defined ones loaded in memory. When activated, macros and conditionals change the status of filled and confirmed slots using predicates such as those presented in section 4.2.

5.1.2 GF grammar for "TownInfo" Voice Programming

The GF grammar for information seeking was extended to cover voice-programming for macros and conditionals in information seeking, based on the WoZ data collection. The user can define and activate macros and conditionals using phrases such as "romantic night" but also "program two", etc.

The following are examples of GF parsing user input, with the corresponding semantic output:

```
> p "when i say french make it expensive"

UttRule (UttRulePhrs80 When_i_say (Bpm_generalTypeRule_13
(Bpm_town_info_restaurants_cuisine_french_Type_Rule
Bpm_town_info_restaurants_cuisine_french)) Make_it
(Bpm_generalTypeRule_16 (Bpm_town_info_restaurant_price_expensive_Type_Rule
Bpm_town_info_restaurant_price_expensive)))

> p "when i say romantic night it means an expensive french restaurant"

UttRule (UttRulePhrs79 When_i_say Romantic_night It_means An
(Bpm_generalTypeRule_16 (Bpm_town_info_bars_price_expensive_Type_Rule
Bpm_town_info_bars_price_expensive)) (Bpm_generalTypeRule_13
(Bpm_town_info_restaurants_cuisine_french_Type_Rule
Bpm_town_info_restaurants_cuisine_french)) (Bpm_spotting_restaurants_1
spot_restaurants_restaurant))
```

```
> p "when i say program one i mean  a cheap central hotel"

UttRule (UttRulePhrs79 When_i_say Program_one I_mean A
(Bpm_generalTypeRule_3 (Bpm_cheap_Type_Rule Bpm_price_range_cheap))
(Bpm_generalTypeRule_8 (Bpm_central_Type_Rule
Bpm_location_type_central)) (Bpm_spotting_hotels_1 spot_hotels_hotel))


> p "program three"

UttRule (UttRulePhrs78 Program_three)
```

The GF grammar was compiled to a speech recognition Language Model used with the ATK recogniser in the prototype system. It could also be compiled to a Nuance grammar [BPL+06].

5.2 Voice Programming for in-car devices

To this system, we also added the capability to define macros and conditionals and also event-driven programs over *devices*, using programming by voice. For development and testing, we used the TALK Display Agent to construct a simple GUI showing the state of several in-car devices.

The currently supported devices are: windows, roof, air conditioning, satellite navigation, and radio, as shown in figure 5.1. The currently supported external events are wind, rain, and low/high temperature.

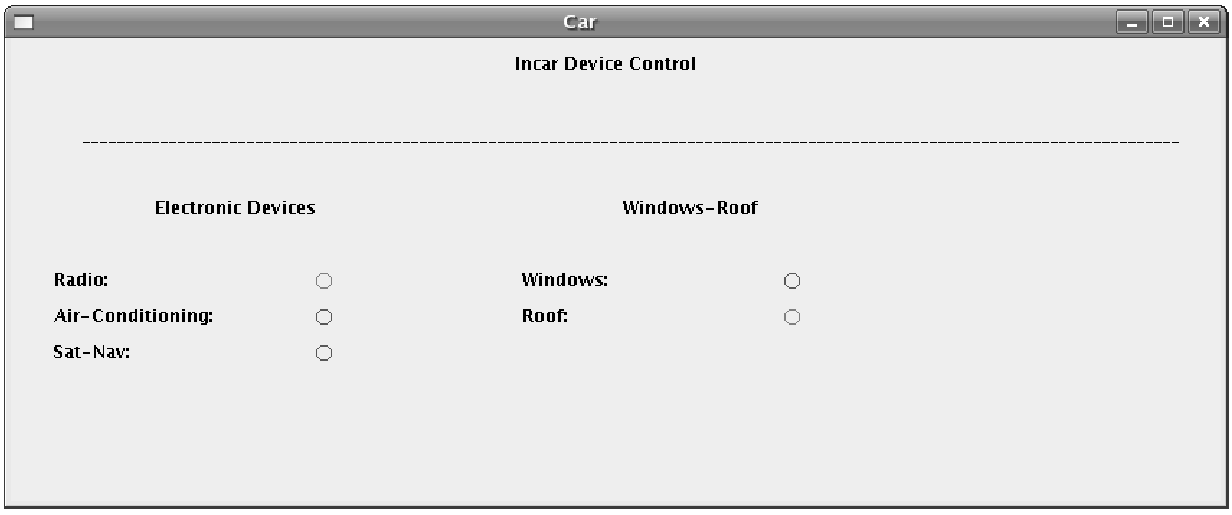

Figure 5.1: The in-car device control GUI

The user programmable in-car device "modes" are: music mode, quiet mode, night mode, and cruise mode, as well as the standard "program one" , "program two", etc.

5.2.1 Dialogue Management for Event-driven programs

Event-driven programs are defined in the same way as conditionals. However, their execution is driven by events and not by user requests. An event is an external action, e.g. a sensor detecting low temperature. There are two methods to implement event-driven programs: polling and interrupts. In polling the system loops, checking if an event has been detected or not, whereas in the interrupt mode the system will stop and wait, or perform a different task until an event is detected. Once an event is detected it will interrupt the current process and the system will carry out the task associated with the detected event. In our case, everything is implemented in polling mode.

To incorporate this feature, we have slightly modified the DIPPER dialogue manager. As previously explained, once the user defines a program it is stored in the Information State (see section 4.1). Now when the user enters the normal mode of information-seeking about hotels, bars, and restaurants, the dialogue manager DIPPER [BKLO03] performs the following actions: First, it can randomly change the status of an event or decide not to change any events at all in the beginning of the dialogue. This is to simulate possible environmental events that could happen in a real system. Then DIPPER loops (polling) through all supported events, checking if an event has been detected. DIPPER then reads all information stored in the macros, conditionals, and event-driven programs fields and transforms them into Prolog clauses that are asserted in memory. Moreover, word patterns that will be used by the parser are also formed and loaded. If a detected event is associated with an event-driven program then other events/actions will be triggered.

Thus in the example of section 4.1 the 'windy' event will cause the events/actions 'close the windows' and 'close the roof' to be performed. DIPPER will then inform the user about the event that has taken place and the events/actions that it has triggered. In addition, the handling of macros and conditionals over devices is the same as for in-car services. That is, every time the parser processes a user input it searches these new Prolog clauses and not only takes into account the standard word patterns but also the user-defined ones loaded in memory. This is done so that the system can detect if the user's request is associated with stored macros or conditionals. Activated macros and conditionals change the status of filled and confirmed slots using predicates such as those presented in section 4.2.

5.2.2 GF grammar and speech recognition for in-car device Voice Programming

For in-car device control and voice programming the system supports the following types of utterances, based on the WoZ data collection (replacing names of in-home devices with in-car devices):

Commands:

"Turn on the sat nav, turn off the radio, and open the roof."
"Open the roof and close the windows."
"I want music mode."
"Quiet mode."

Voice Programming:

“When I say music mode I mean close the windows and turn on the radio”
“When I say quiet mode it means turn off the air conditioning and turn off the radio”
”When there is rain then close the roof and close the windows and turn off the radio”

We extended the GF grammar to handle such user utterances, for example:

```
> p "when i say music  mode i mean turn off the sat nav and close the windows"

UttRule (UttRulePhrs86 When_i_say Music_mode I_mean Turn_off The
(Bpm_spotting_sat_nav_1 spot_devices_sat_nav) And Close The
(Bpm_spotting_windows_1 spot_devices_windows))


> p "when i say music mode i mean turn off the sat nav and open the
     roof and turn on the radio"

UttRule (UttRulePhrs87 When_i_say Music_mode I_mean Turn_off The
    (Bpm_spotting_sat_nav_1 spot_devices_sat_nav) And Open The
     (Bpm_spotting_roof_1 spot_devices_roof) And Turn_on The
      (Bpm_spotting_radio_1 spot_devices_radio))


> p "when i say quiet mode it means turn off the sat nav and close the windows"

UttRule (UttRulePhrs86 When_i_say Quiet_mode It_means Turn_off The
(Bpm_spotting_sat_nav_1 spot_devices_sat_nav) And Close The
(Bpm_spotting_windows_1 spot_devices_windows))


> p "open the windows and close the roof"

UttRule (UttRulePhrs85 Open The (Bpm_spotting_windows_1
spot_devices_windows) And Close The (Bpm_spotting_roof_1
spot_devices_roof))

> p "turn off the radio and turn on the sat nav"

UttRule (UttRulePhrs85 Turn_off The (Bpm_spotting_radio_1
spot_devices_radio) And Turn_on The (Bpm_spotting_sat_nav_1
spot_devices_sat_nav))

> p "when i say quiet mode it means  turn off the radio"

 UttRule (UttRulePhrs84 When_i_say Quiet_mode It_means Turn_on The
 (Bpm_spotting_radio_1 spot_devices_radio))
```

```
> p "when there is  rain then close the roof and close the windows and turn off the radio"

UttRule (UttRulePhrs87 When_there_is Rain Then Close The
(Bpm_spotting_roof_1 spot_devices_roof) And Close The
(Bpm_spotting_windows_1 spot_devices_windows) And Turn_off The
(Bpm_spotting_radio_1 spot_devices_radio))



>  p "if it is hot then  turn on  the air conditioning and open the windows"

UttRule (UttRulePhrs86 If_it_is Hot Then Turn_on The
(Bpm_spotting_air_conditioning_1 spot_devices_air_conditioning) And
Open The (Bpm_spotting_windows_1 spot_devices_windows))


> p "turn on the radio"

> UttRule (UttRulePhrs71 Turn_on The (Bpm_spotting_radio_1 spot_devices_radio))
```

The GF grammar was compiled to a speech recognition Language Model used with the ATK recogniser in the prototype system.

5.3 Example dialogue

The system now supports the following sort of dialogues:

- USER: open the roof and turn off the radio
- SYSTEM: ok [device states change on GUI]
- SYSTEM: I have opened the roof
- SYSTEM: I have turned off the radio
- SYSTEM: how may I help you?
- USER: when I say romantic night it means an expensive French restaurant
- SYSTEM: ok, your program has been saved
- USER: when I say quiet mode it means close the windows and close the roof and turn off the radio
- SYSTEM: ok, your program has been saved
- USER: if it is hot then turn on the air conditioning and open the windows
- SYSTEM: ok, your program has been saved

- USER: when I say cruise mode it means turn on the radio and open the roof
- SYSTEM: ok, your program has been saved
- USER: I want quiet mode
- SYSTEM: ok [device states change on GUI]
- SYSTEM: I have closed the windows
- SYSTEM: I have closed the roof
- SYSTEM: the radio is already off
- SYSTEM: how may I help you?
- USER: ok, how about a romantic night?
- SYSTEM: Ok, an expensive French restaurant, Do you want something central?
- USER: yes please
- SYSTEM: Chez Sergu is a fabulous French restaurant in the centre of town...

Note that the system always implicitly confirms activated macros, conditionals, and event-driven programs, as discussed in section 2.2.2, for example in the system utterance "Ok, an expensive french restaurant. Do you want something central?" in the above dialogue.

5.4 Summary

This chapter presented the "in-car" TownInfo system of D4.2 [LGS05] and [LGHS06] which was extended to allow voice programming of both in-car devices and services [GL06], using macros, conditionals, and event driven programs. The system uses components from the TALK project infrastructure [BPL+06] and also exploits the strengths of the Information State Update approach to dialogue in its generic treatment of voice programming functions.

To our knowledge this is the first dialogue system to develop voice programming functionalities.

Chapter 6

Voice Programming using ontologies in the Linguamatics in-home system

6.1 Introduction

In a dynamically reconfigurable, ontology-based dialogue system, the ontology is no longer a static data structure storing fixed domain knowledge, but a dynamic data structure that can be changed by external tasks during the progress of a dialogue session. For example, in the Linguamatics Interaction Manager (LIM), external tasks can dynamically change the ontology by sending messages to the dialogue manager. Possible changes to the ontology include the addition and deletion of devices and services. A macro program is an example of a new service where a single command executes a set of commands over existing devices.

Given the ability to add new services on the fly, it is natural to treat voice programming in a similar fashion by allowing new user-defined services as well as new services defined by external tasks.

In this chapter we first motivate the use of voice programming in our application domain of home information and control. We then discuss different ways of defining voice programs, and the requirements for recognition and interpretation. We then describe how voice programming has been implemented in the Linguamatics in-home showcase.

6.2 Voice Programming and home automation

A more automated home environment should have benefits both for the comfort of individuals in the home, and in reducing the costs of heating and lighting. However, it is recognised that programming a home is a difficult task using current technology. There are two main approaches to dealing with this: firstly by making interfaces more intuitive and easier to use (of which voice programming is a part), and self-learning systems which aim to learn the behaviour of the inhabitants of the home, and adapt the home's behaviour to predict their needs [Moz05]. The latter approach attempts to hide complexity from the user, but there will still be a need for users to over-ride the automated behaviour. Mozer's suggestion that users should adapt to their automated home by behaving predictably might not suit everyone.

As well as programming heating and lighting, another area which is important is programming as part

of the configuration of the home. One of the potential benefits of home automation is that it can allow for more modular devices. For example, one set of quality hi-fi speakers, rather than separate speakers for the audio system and the television system. These devices need to be linked together. To drive down the cost of the automated home the ability to configure the devices should be accessible to users, not just experts. For example, the user should be able to say "route audio output from the lounge tv to the lounge hi-fi speakers". They may also need to configure the remote e.g."when the audio volume is turned down, turn down audio volume for lounge hi-fi speakers". Users should also be able to define virtual devices by combining simpler devices e.g. take a computer timer and a cooker to provide a cooker with a timer.

6.3 Dialogue Management for Multi-Step Voice Programming

In the earlier examples we mainly concentrated on programs which are defined in a single step such as the following examples:

- USER: When the phone rings mute all the audio equipment
- USER: When I say mode one it means close the windows and close the roof and turn off the radio

The same effect can also be achieved using *multi-step* dialogues, e.g.

- USER: I'd like to create a program
- SYSTEM: ok, when would this program be activated?
- USER: when the telephone rings
- SYSTEM: what would you like to occur when the program is triggered?
- USER: mute all the audio equipment
- SYSTEM: ok, your program has been stored

It is also possible to imagine more complex exchanges where arbitrarily many actions are defined by a user:

- USER: I'd like to create a program
- SYSTEM: ok, when would this program be activated?
- USER: when I say "mode one"
- SYSTEM: any other condition?
- USER: no
- SYSTEM: what would you like to occur when the program is triggered?
- USER: close the windows
- SYSTEM: anything else?

- USER: close the roof
- SYSTEM: anything else?
- USER: turn off the radio?
- SYSTEM: anything else?
- USER: no
- SYSTEM: ok, your program has been stored so that when you say “mode one” the following will occur: “windows will close, roof will close, radio will be turned off”.

Ideally a system would support both single-step and multi-step definitions of programs. However, there are issues concerning recognition and interpretation of voice programs which we will discuss in the next section.

6.4 Recognising and Interpreting Voice Programs

In many dialogue interactions, there is a certain amount of redundancy, with users repeating parts of the request. For example, consider the following exchange:

- SYSTEM: Where would you like to go?
- USER: I would like to go to Paris

In the user’s utterance, only the word “Paris” is actually critical to getting a correct interpretation.

Voice programming however tends to be very different, with little redundancy. For example, consider the following utterance:

- USER: When the phone rings mute all the audio equipment

All the words here except “the” and possibly “equipment” are critical for the interpretation. This means that any misrecognition is much more likely to cause an incorrect interpretation.

Using a grammar-based approach to language modelling (as in the extended TownInfo system) will shoe-horn utterances into appropriate templates for voice programming, and will tend to work well if users can remember the appropriate structuring for their utterances. However, there is also a danger of utterances being shoe-horned into the wrong templates (see e.g. [KGR+01] concerning evaluation of grammar-based vs. statistical language modelling in the home control domain).

A statistical language model may provide a better average error rate than a grammar-based approach. However, for both approaches the likelihood of an error occurring somewhere in the utterance will increase with length of utterance. This is going to be critical if the errors occur for non-redundant words.

An added complication with the use of a statistical language model together with fragment parsing is that these utterances contain perfectly well formed fragments. For example “mute all the audio equipment” is a well-formed command which could be uttered in the same context as “When the phone rings mute all the audio equipment”. It is thus necessary to use a strategy which will prefer the interpretation of a longer fragment over an equally plausible smaller fragment.

In conclusion, although defining programs in a single long utterances may be convenient for a user, the ability to back-off to a multi-step treatment of voice programming may be necessary given current speech recognition accuracy.

6.5 Implementation of Voice Programming

The Linguamatics Interaction Manager is used to control human-machine communication. It interprets speech or a mouse click, and responds by speech or by providing a new graphical display, or a combination of the two. The system is designed to be highly reconfigurable to enable its use in dynamic scenarios where the whole task or ontological structure, and the vocabulary, can change.

6.5.1 Adapting the ontology

To allow users to define new concepts on the fly requires the ability to add new concepts to the ontology at run-time, and to define the programs themselves. In the Linguamatics Interaction Manager, the following lines define a new concept, “program_one”.

sem program_one instance pub my_services
define program_one ”program one”
userout program_one ”program one”

The first line tells the dialogue manager to create a new concept “program_one” which will be public and placed under the concept “my_services”. This means that when the user next navigates to the “my_services” concept, “program_one” will now be viewable as a child of “my_services”. The second line defines “program one” to be a synonym for the concept “program_one”. The synonym(s) are used during interpretation. The third line defines “program one” as the term used for output. This is used during language generation, and for labelling icons on screen.

To adapt the ontology to define programs involving multiple changes of state requires an appropriate syntax. In the Linguamatics Interaction Manager the “end_of_line” token is used to separate multiple commands. For example, to close the window and open the roof would be achieved by defining the output of “program_one” to be two commands “close window” and “open roof”. These are prefixed by “car” to tell the router to send the commands to the car manager:

outputfor program_one “car close window” end_of_line “car open roof”

6.5.2 Simple Voice Programming

The LIM system allows users to define simple macro progams by associating a set of predefined names (e.g. “program one”) to a the state of a particular device. In the following example, the user defines “program one” to be equivalent to setting the lounge television to the channel BBC1.

- USER: Add service
- SYSTEM: Which service to add?
- USER: Program

- SYSTEM: Which program?
- USER: program one
- SYSTEM: Which device?
- USER: the lounge television
- SYSTEM: Which setting?
- USER: channel
- SYSTEM: which channel?
- USER: BBC1
- SYSTEM: added program one to your services

This dialogue is a case of macro programming, equivalent to "when I say program one turn the lounge television to BBC1". The change to the ontology is as follows:

```
sem program_one instance pub my_services
define program_one "program one"
userout program_one "program one"
outputfor program_one "house bbc1 lounge_1_tv channel
```

The "outputfor" specification line defines the output for "program_one" when activated. The word "house" is used to route a command or request to the home manager. "bbc1" provides the channel, "lounge_1_tv" specifies the device. "channel" specifies the kind of parameter to be changed (in this case the value "bbc1" can only be a channel, but other values e.g. 50% might be associated with multiple parameters such as brightness or volume).
Once the program is added, it is available to be used. Users can navigate to the "my services" menu, or they can ask for the program to be performed from the top-level menu e.g.

- USER: back to the top
- SYSTEM: which service?
- USER: program one
- SYSTEM: switched the lounge tv to bbc1

The interaction manager also allows users to define event-driven programs. For example, the following dialogue defines an event driven program which turns the kitchen light on when the door is opened:

- SYSTEM: Which program?
- USER: event one
- SYSTEM: Which device?
- USER: kitchen door

- SYSTEM: Open or closed?
- USER: open
- SYSTEM: Which device?
- USER: kitchen light
- SYSTEM: on or off?
- USER: on
- SYSTEM: Event added so that when kitchen door is open , kitchen light is on

It is currently assumed that the event-driven programs are the responsibility of the house manager rather than the interaction manager, so the final step sends a message containing the program to the house. Other architectures would be possible, for example, where the program is still stored in the ontology, but the house manager and dialogue manager share the same ontology as a shared representation of the state of the house.

6.5.3 Recognition and Interpretation

The Linguamatics home control language model uses a statistical approach, with semantics being extracted from fragments of an utterance. Consider an utterance such as the following:

> "Turn on the light in the hall when the front-door is opened."

The semantic interpretation provides the largest coherent composition of semantic units. However, currently syntax (including morphology) is not taken into account, so the system would not be able to distinguish the utterance above from:

> "Open the front-door when the hall light is turned on."

The strategy of relying on semantics alone was intended as a stop-gap measure, but has been more successful that expected. However, voice programming does provide good examples where the semantics of the individual words alone is not enough to elucidate the meaning of the whole. In the future we intend to incorporate both syntactic and semantic processing, but running in parallel so that semantic processing can still win relative to a syntactically well formed, but meaningless interpretation. In the meantime we have limited voice programming to the simple multi-step examples above.

6.6 Conclusion

In this chapter we have described examples of using voice programming with the Linguamatics Interaction Manager in the home environment by storing programs in the ontology. Although the examples are somewhat limited, this is largely due to the way interpretation has been implemented in the Linguamatics showcase. The general approach seems successful, and, for systems that can refine their ontologies on the fly, storing programs in the ontology seems a valid alternative to storing programs in the Information

State. The adoption of one approach or the other should depend on more general distinctions in the use of ontologies vs. information states for different kinds of knowledge. In the Linguamatics system, for example, the approach is that the Information State only contains information about the current interaction. Changes that occur during a dialogue, but are not related directly to the interaction are stored in the ontology.

Chapter 7

Conclusion

We have presented generic approaches to voice programming, based on the ISU approach to dialogue, and our implementations of voice-programming for devices and services, for both in-home and in-car domains.

Voice programming is a powerful idea, in that it allows users to have direct and explicit control over the voice commands that they can use in a spoken dialogue interface. Similar to users defining keyboard macros or properties of a GUI, voice programming offers the user a new type of adaptivity for spoken dialogue systems. In addition, the shorter and less complex utterances enabled by voice programs are likely to improve speech recognition, and thus system robustness.

We added to the TALK in-car system [LGS05, LGHS06] features for programming using macros, conditionals, and event-driven programs. This voice programming demonstration system uses software infrastructure developed in the TALK project, for example we used GF (UGOT, WP1), ATK (UCAM, WP1) and the database and display agents (DFKI, WP5) [BPL+06].

We also added simple examples of macros and event-driven programs to the in-home application for the Linguamatics Interaction Manager. This showcase emphasises dynamic reconfiguration using ontologies, and has been described in detail in the deliverables D2.1 and D2.2.

We first showed that there is a number of natural language constructs that can be used in voice programming, such as macros, conditionals, loops, and event-driven programs. We then described the WoZ data collection and analysis that we performed for Voice Programming.

Next we presented the technical challenges that have arisen in this task, for example generic ways of extending Information States, adding new Update Rules to support voice programming, and the issues of unknown words and extending the existing language models and parser appropriately, based on the results of the WoZ experiments.

With respect to "state of the art", we noted that, as far as we are aware, no dialogue system has previously implemented voice programming functionalities. We discussed the related work of Metafor [LL05b, LL05a], IBL [LKB+02] and GSI [TGS06], each of which addresses different issues in using Natural Language in programming of various types.

Note that the approaches developed here are quite general and portable. For example, the Update Rules and Information States that we have developed can be re-used in any domain. All that is required is to define the trigger phrases that users can employ for their macros and conditionals or event-driven programs, and this is localized to the grammar module.

7.1 Future Challenges

The outstanding future challenge in this area is to allow the user to define trigger words or phrases that are currently unknown to the system. This problem has been approached in *text-based* dialogue, using clarification questions to specify meanings of unknown words [Pur02]. For *spoken* dialogue systems the new difficulty here, clearly, is that the recogniser Language Model (LM) may not contain the user's new trigger word or phrase, so it could be in principle impossible to add it using voice alone. One possible solution is to allow users to spell words that are not in the current LM. Another would be to enter special subdialogues for defining new trigger phrases, where a large "bag of words" LM, restricted to sequences of up to two or three words, is used specifically for recognising novel phrases.

Future system development would include providing the user with more advanced functionality. For example supporting more complex syntax, e.g. combinations of "and" and "or" operators, loops, macros that call other macros, etc. Moreover, it would be very interesting to investigate the issue of dependency between macros and conditionals as discussed in section 2.2.2, and the issue of how voice programs should update parts of the dialogue context such as salient NPs for pronoun resolution. *Multimodal* versions of voice programming, allowing users to mix graphical and verbal triggers and commands, could also be useful for certain applications (e.g. CAD interfaces).

We see no problems in principle with such extensions, based on the generic ideas and techniques we have developed in this deliverable.

Bibliography

[BKLO03] Johan Bos, Ewan Klein, Oliver Lemon, and Tetsushi Oka. DIPPER: Description and Formalisation of an Information-State Update Dialogue System Architecture. In *4th SIGdial Workshop on Discourse and Dialogue*, Sapporo, 2003.

[BPL+06] Tilman Becker, Peter Poller, Staffan Larsson, Oliver Lemon, Guillermo Pérez, and Karl Weilhammer. D5.1: Software infrastructure. Technical report, TALK project, 2006.

[Cyp93] Allen Cypher. *Watch What I Do*. MIT Press, 1993.

[GL06] Kallirroi Georgila and Oliver Lemon. Programming by Voice (demonstration system): enhancing adaptivity and robustness of spoken dialogue systems. In *Proceedings of Brandial, the 10th SemDial Workshop on the Semantics and Pragmatics of Dialogue, (demonstration systems)*, 2006.

[KGR+01] S. Knight, G. Gorrell, M. Rayner, D. Milward, R. Koeling, and I Lewin. Comparing grammar-based and robust approaches to speech understanding: a case study. In *Proc. Eurospeech 2001, 7th European Conference on Speech Communication and Technology*, Aalborg, Denmark, 2001.

[LGHS06] Oliver Lemon, Kallirroi Georgila, James Henderson, and Matthew Stuttle. An ISU dialogue system exhibiting reinforcement learning of dialogue policies: generic slot-filling in the TALK in-car system. In *Proceedings of EACL*, 2006.

[LGS05] Oliver Lemon, Kallirroi Georgila, and Matthew Stuttle. D4.2: Showcase exhibiting Reinforcement Learning for dialogue strategies in the in-car domain. Technical report, TALK Project, 2005.

[LKB+02] S. Lauria, T. Kyriacou, G. Bugmann, J. Bos, and E. Klein. Converting natural language route instructions into robot executable procedures. In *Proc. of the 2002 IEEE International Workshop on Robot and Human Interactive Communication (Roman'02), Berlin, Germany, pp. 223-228.*, 2002.

[LL05a] Hugo Liu and Henry Lieberman. *Feasibility Studies for Programming in Natural Language*. Kluwer, 2005.

[LL05b] Hugo Liu and Henry Lieberman. Metafor: Visualizing stories as code. In *Proceedings of the ACM International Conference on Intelligent User Interfaces, IUI*. ACM, 2005.

[MAB+06] David Milward, Gabriel Amores, Nate Blaylock, Staffan Larsson, Peter Ljunglof, Pilar Manchon, and Guillermo Perez. D2.2: Dynamic multimodal interface reconfiguration. Technical report, TALK project, 2006.

[Moz05] M. C. Mozer. Lessons from an adaptive house. In D. Cook and R. Das, editors, *Smart environments: Technologies, protocols, and applications*, pages 273–294. Hoboken, NJ: J. Wiley & Sons., 2005.

[Pur02] Matthew Purver. Processing unknown words in a dialogue system. In *Proceedings of the 3rd SIGdial Workshop on Discourse and Dialogue*, pages 174–183, Philadelphia, July 2002. Association for Computational Linguistics.

[Ran04] A. Ranta. Grammatical framework. a type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189, 2004.

[TGS06] E. Tse, S. Greenberg, and C. Sheen. GSI Demo: Multiuser Gesture/Speech Interaction over Digital Tables Wrapping Single User Applications. In *ICMI*, pages 76–83, 2006.

[YST+06] Steve Young, Jost Schatzmann, Blaise Thomson, Karl Weilhammer, and Hui Ye. D4.4: POMDP-based System. Technical report, TALK Project, 2006.

Appendix A

Software delivered for TownInfo System

For the Voice Programming system described in chapter 5, the software accompanying this deliverable consists of:

- DIPPER information state file
- DIPPER update rules file
- DIPPER resources file
- HTK lattice and dictionary files
- Display Agent for in-car devices

For a full working system, these items should replace the corresponding files in the TALK TownInfo system of deliverable 4.2 [LGS05].

Appendix B

Software delivered for In-Home System

The Linguamatics Interaction Manager consists of the following modules:

1. Java router
2. C executable
3. The following configuration files:
   (a) Specification of path information for temporary files
   (b) Communication strings for sending messages to the recogniser, house etc.
   (c) Formatting options for graphical output (including definition of XML tags if appropriate)
   (d) Escape options for the grammar
   (e) Ontology

The particular modules changed for Voice Programming are the following:

1. C executable
2. Ontology