Software illustrating a unified approach to multimodality and multilinguality in the in-home domain

Stina Ericsson (editor), Gabriel Amores, Björn Bringert, Håkan Burden, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, Mikael Sandin

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable 1.6

December 22, 2006

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	December 22, 2006
Deliverable number	1.6
Deliverable title	Software illustrating a unified approach to multimodality and multilinguality in the in-home domain
Type	Report
Status & version	Final December 22, 2006
Number of pages	121 (excluding front matter)
Contributing WP	1
WP/Task responsible	UGOT
Other contributors	All partners
Author(s)	Stina Ericsson (editor), Gabriel Amores, Björn Bringert, Håkan Burden, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, Mikael Sandin
EC Project Officer	Evangelia Markidou
Keywords	Multimodality, Multilinguality, In-Home Dialogue Systems

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 1

1 Introduction **2**

2 Multimodality and Multilinguality in GoDiS **4**

2.1 Dialogue management for multimodality and multilinguality 4

2.1.1 Dialogue management in GoDiS 4

2.1.2 Multimodality in GoDiS 5

2.1.3 Multilinguality in GoDiS 11

2.2 GF grammar work related to GoDiS applications 11

2.2.1 Grammars for Gottis 13

2.2.2 Grammars common to all GoDiS applications 15

2.2.3 Grammars for GoTGoDiS 17

2.2.4 Grammars for AgendaTalk 18

2.2.5 Grammars for DJ-GoDiS 20

2.2.6 Grammars for GoDiS-deLux 21

2.3 GoDiS applications demonstrating multimodality and multilinguality 23

2.3.1 Introduction 23

2.3.2 Gottis 23

2.3.3 GoTGoDiS 26

2.3.4 AgendaTalk 34

2.3.5 DJ-GoDiS 50

2.3.6 GoDiS-deLux 60

2.4 Conclusion 68

3 Multimodality and Multilinguality in the Linguamatics Interaction Manager **70**

3.1 Introduction 70

3.2 System Summary 70

3.3 Issues Addressed 71

3.4 Multilinguality 71

3.5 Multimodality 71

3.6 Speech Recognition 72

3.7 Multimodal Output . . . 73
3.8 Moving to a unified approach to multimodality and multilinguality . . . 73
3.9 The Home Domain Showcase . . . 73
3.10 Conclusion . . . 75

4 Multimodality and Multilinguality in MIMUS 76
4.1 Introduction . . . 76
4.2 Scenario . . . 76
4.2.1 WoZ Experiments . . . 77
4.3 Infrastructure . . . 78
4.4 The ISU Approach in MIMUS . . . 81
4.4.1 DTAC Information States . . . 81
4.4.2 Multimodal DTAC structure . . . 82
4.4.3 Updating the Information State in MIMUS . . . 83
4.5 Multimodality and Multilinguality in MIMUS . . . 84
4.5.1 Integrating OWL in MIMUS . . . 84
4.5.2 From OWL to the House Layout . . . 87
4.5.3 From Ontologies to Grammars: OWL2Gra . . . 87
4.5.4 Multilinguality in MIMUS . . . 93
4.5.5 Multimodality in MIMUS . . . 96
4.5.6 Multimodal Presentation in MIMUS . . . 100
4.6 Dialogue Examples . . . 101
4.7 Conclusion . . . 104

5 Conclusion 106
5.1 Advantages over current state-of-the-art . . . 106
5.2 Advantages of the ISU approach . . . 107
5.3 Implementation of research in the showcases . . . 108

Bibliography 109

A Software Library 114
A.1 Software for GoDiS . . . 114
A.1.1 Grammars . . . 114
A.1.2 Applications . . . 118
A.2 Software for the Linguamatics Interaction Manager . . . 119
A.3 Software for MIMUS . . . 119
A.3.1 Root Directory: Batch Files . . . 119
A.3.2 VRM . . . 119
A.3.3 Talking Head . . . 119
A.3.4 Ontology . . . 120

A.3.5 MMInputPool 120
A.3.6 MimusCore 120
A.3.7 Merlin 120
A.3.8 HomeSetup 120
A.3.9 jDeviceManagerAgent 120
A.3.10 jKManagerAgent 120
A.3.11 jDisplayAgent 120
A.3.12 jMenuAgent, jMP3Agent, jTelephoneAgent 121

Summary

This deliverable presents work on Task 1.6 in the TALK project. The research task focuses on multimodality and multilinguality for the Information State Update (ISU) approach in the in-home domain, with particular attention paid to a unified approach to multimodality and multilinguality. Such a unified approach involves contributions in different modalities and different languages as concrete realisations of common abstract representations, creating coherent, powerful, and efficient dialogue systems that allow rapid development and porting.

The deliverable showcases three systems – GoDiS, the Linguamatics Interaction Manager, and MIMUS. GoDiS is developed for four different applications, the control of lights, an mp3 player, a calendar, and public transportation information. The Linguamatics Interaction Manager allows the control of a large number of different devices in the home, such as lights and television sets. MIMUS includes a talking head, and lets the user control different devices in the home. GoDiS and MIMUS are both multimodal and multilingual, for input as well as for output. The Linguamatics Interaction Manager is also multimodal for both input and output. The integrated approach to multimodality and multilinguality for GoDiS has been developed by connecting GoDiS to the Grammatical Framework (GF), which supports rapid porting of GoDiS to new domains, modalities, and languages. For instance, one of the GoDiS applications has been developed for seven different languages, of which one is the non-Indo-European language Finnish. Languages can be changed on the fly, in the middle of a dialogue, in both MIMUS and GoDiS. The two systems together show two different ways of achieving language change: either by pressing a button for the desired language, or by naming the language using speech. During language change, the Information State maintains language-independent knowledge of the current state, so it is possible to switch languages in the middle of a task.

Chapter 1

Introduction

An in-home interactive system that allows maximal flexibility and user-friendliness, and is accessible to a wide variety of users, is one that handles several different languages, and that enables communication in the modalities – speech, graphics, gestures, and so on – most natural to the user and the interactive activity in question. Such an interactive system does all of this in an efficient and powerful way, not only from the perspective of the end user, but, of equally high importance to our concerns, from a design and implementation perspective.

One way in which this can be achieved is through the unification of multimodality and multilinguality in a single framework, such as through the use of a single abstract representation for languages and modalities. In this way, the same information can be used to generate concrete representations, both in various natural languages such as English, Spanish, and Swedish, and in different modalities. From an interpretation perspective, linguistic and non-linguistic information can be unified into an abstract representation that can be handled by the same dialogue management and other components in the system. Thus, unified abstract representations allow for a coherent view of multilinguality and multimodality, giving the dialogue system designer and implementer a very powerful environment, and the end user an integrated system.

Coupled with an information state update (ISU) approach, a unified approach to multimodality and multilinguality creates a particularly potent approach. The ISU approach utilises structured information states to keep track of dialogue information. These information states can be read and updated by several different modules which access precisely the information that they need. This enables a modular architecture which allows generic solutions for dialogue technology. For example,

- different language modules can interact with essentially similar information states, enabling rapid porting of dialogues systems from one language to another and the creation of multilingual dialogue systems
- coding of dialogue behaviour is supported independent of language and domain, thus allowing for the rapid porting of dialogue systems to different domains
- the use of structured information states allows straightforward implementation of flexible dialogue systems which can access and modify information in the information state in varying orders and with varying means

Treating multimodality as additional “languages” in the system thus does not require a significant rebuilding of an ISU-based unimodal system since the ISU technology already allows for the modular addition of

new languages. Among other things, we show in this deliverable how a unimodal system can be rapidly extended to a corresponding multimodal system.

We showcase a variety of systems, all implementing our research on multimodality and multilinguality in the in-home domain and with respect to the ISU approach. Presented in alphabetical order, we discuss GoDiS, the Linguamatics Interaction Manager, and MIMUS. GoDiS, in Chapter 2, is concerned with the integration of the Grammatical Framework (GF), for the construction of grammars, with TrindiKit and GoDiS, for the ISU approach to dialogue. Four different GoDiS applications are showcased, showing multimodality and multilinguality in relation to the same abstract representation, as well as rapid prototyping for new languages and new domains, and advanced dialogue management.

The Linguamatics Interaction Manager, in Chapter 3, focuses on domain reconfigurability, and explores the relationship between reconfigurability on the one hand, and multimodality and multilinguality on the other. Domain reconfigurability and multimodality are both implemented in the showcased system, and a unified approach to all three of reconfigurability, multimodality, and multilinguality is explored from a theoretical viewpoint.

MIMUS, in Chapter 4, explores a unified approach to multimodality and multilinguality using OWL ontologies. Different approaches to multimodal fusion of user input are investigated and evaluated, and several Wizard-of-Oz studies have been carried out, the results of which have fed into the multilingual and multimodal development of MIMUS.

Chapter 2

Multimodality and Multilinguality in GoDiS

This chapter addresses the issue of a unified approach to multimodality and multilinguality in GoDiS using the Grammatical Framework (GF). The issue is tackled in relation to four GoDiS in-home applications: GoTGoDiS for information about public transport, AgendaTalk for interaction with an electronic calendar, DJ-GoDiS for the control of an mp3 player, and GoDiS-deLux for controlling lights in the home. The chapter also includes Gottis, a non-GoDiS-based system implemented as a form of baseline system for multimodality and multilinguality in GF. This system was converted into the GoDiS application GoTGoDiS to illustrate the dialogue management behaviour of GoDiS.

Section 2.1 tackles general issues that concern dialogue management for multimodality and multilinguality in the context of GoDiS. Section 2.2 then traces the development of GF grammars for GoDiS, detailing multilingual and multimodal grammars for Gottis and the GoDiS applications, and Section 2.3 describes the four GoDiS applications and how they illustrate multilinguality and multimodality.

2.1 Dialogue management for multimodality and multilinguality

This section briefly describes dialogue management aspects of multimodality and multilinguality as implemented in GoDiS. Dialogue management in the ISU approach has three components: information state, update rules, and overall system architecture and control. Dialogue management modifications to enable multimodality and multilinguality will thus be described in terms of extended information states, update rule modifications, and architectural/control modifications.

2.1.1 Dialogue management in GoDiS

This section offers a very brief introduction to dialogue management in GoDiS in relation to current state of the art. For a more detailed description of GoDiS, see [13] and [12]. GoDiS implements Issue-based Dialogue Management, a general theory of dialogue processing based on the notion of dialogue as, essentially, raising and addressing questions (or "issues"). The current version of GoDiS is implemented

in the Information State Update approach using TrindiKit4.

One of the main challenges addressed in this WP is to combine the existing advanced flexible dialogue management and rapid prototyping capabilities of GODIS with multimodal and multilingual dialogue.

The domain-independent GODIS Dialogue Move Engine (DME) provides general solutions to several generic problems of dialogue processing: grounding, feedback, clarification, multiple simultaneous tasks, sharing information between tasks, user initiative, belief revision, and more. In current industrial state-of-the-art dialogue system platforms such as VoiceXML, these generic problems have to be addressed individually in each new application, while in GODIS they are solved by the domain-independent DME. Also, the dialogue management capabilities of GODIS go beyond VoiceXML in several aspects, such as grounding on multiple levels, dealing with multiple simultaneous tasks initiated by the user, belief revision, and plan recognition (dependent accommodation).

The modular structure of GODIS clearly separates dialogue management, overall system control, and domain-specific resources. In combination with general solutions to general dialogue management problems, this enables rapid, and thus cheap, prototyping of new applications. The main application-specific resources are domain knowledge (ontologies and dialogue plans) and GF grammars. The latter are compiled into grammar formats or SLMs which can be used for ASR. Regarding both of these components, GODIS uses pre-existing methods and resources to minimise the work needed for developing and localising new applications. Dialogue plans can be based on existing menu interfaces, which thus provide basic dialogue designs while allowing for advanced flexible multimodal dialogue. Using a generic GUI for graphical interaction minimises the need for developing new GUIs for new applications. Finally, using GF resource grammars and automatic ASR grammar generation from GF grammars minimises the work needed for localising applications to new languages.

2.1.2 Multimodality in GODIS

This section describes multimodality in GODIS. First, we briefly review the Multimodal Menu-based Dialogue (MMD) approach implemented in several GODIS applications. We then describe how the GODIS information state (IS) has been extended to handle multimodal interaction. Next, we outline how GODIS has been modified for asynchronous control to adequately deal with the complexities of multimodal interaction. We also describe DynGUI, a generic MMD interaction GUI used by several applications, as well as the use of application-specific GUIs for multimodal interaction. Finally, we explain how GODIS deals with multimodality across multiple domains.

Multimodal Menu-Based Dialogue (MMD)

Three of the UGOT TALK applications (GOTGODIS, GODIS-DELUX and DJ-GODIS) are based on an approach to multimodality, developed in the TALK project, referred to as Multimodal Menu-based Dialogue (MMD). A more detailed description of this approach can be found in D2.1 [18].

By converting existing graphical menu-based interfaces into dialogue plans, GODIS applications can be built which use the same basic menu structure but in addition allows flexible spoken dialogue interaction. By keeping graphical and spoken interaction in sync, the MMD approach allows the user free choice of modality as well as mixing modalities. The MMD approach has been used in the GODIS-DELUX, DJ-GODIS and GOTGODIS applications. AGENDATALK has not been built with the MMD approach as it does not use a menu-based interface. However, AGENDATALKmakes use of many of the features

described in this section.
Features of MMD interaction include the following:

- MMD GUI interface can be used ”as usual” (no speech)
- Application can be controlled using speech only
- Modalities can be freely mixed and changed at any point
- A novice can follow the menu structure step-by-step to learn application capabilities, and an expert user can bypass menus by asking questions or giving requests directly, or by just providing some piece of information relevant to her goals (in which case the system will accommodate the user's goal or ask a clarification question)

Extended information state for multimodality

This section describes how the GODIS information state (TIS) has been extended to deal with multimodal utterance interpretation and generation.

Utterance interpretation in GODIS can be regarded as a function from TIS variable INPUT to TIS variables LATEST_MOVES and LATEST_SPEAKER.
Prior to TALK, GODIS used simple representations of utterances and utterance interpretations.

- utterances represented as a string of text -
  INPUT **:** String
- utterance interpretations represented as an open queue of dialogue moves -
  LATEST_MOVES **:** Oqueue(Move)
- non-integrated moves represented as an open queue of dialogue moves -
  IS/PRIVATE/NIM **:** Oqueue(Move)
- interpretation of latest utterance represented as a record indicating speaker and a set of moves -
  IS/SHARED/LU **:** $\begin{bmatrix} \text{SPEAKER} & : & \text{Participant} \\ \text{MOVES} & : & \text{Set(Move)} \end{bmatrix}$
- interpretation of previous utterance as an open queue of parts of speaker and move -
  IS/PRIVATE/NIM **:** Oqueue(Pair(Participant,Move))

However, these representations are insufficient to deal with multimodality, where moves can be realised using different modalities. To improve feedback in multimodal interaction, GODIS has also been improved by assigning ASR score to each individual move, and by allowing increased system reconfigurability by running several applications simultaneously (described in D2.2 [19]). Taken together, these developments require a more complex utterance representations, and we have opted for representing utterances and interpretations as records containing several fields. One advantage of this record-based representation is that it can be extended without disrupting or requiring modifications to existing dialogue processing.
The INPUT variable represents utterances observed by the system as a queue of records containing three fields. The value of the TOKEN field is a string of text, regarded by the grammar as a complete utterance.

The string can contain both spoken words and GUI input. The MODALITIES field is the set of modalities used to express that move (either speech or gui), and SCORE is the ASR score (in case the speech modality was used; otherwise, it is set to the maximum value 1.0[1]):

(1) INPUT : Queue([TOKEN : String; MODALITIES : Set(Modality); SCORE : Real])

In LATEST_MOVES, the MOVE field is a GODIS dialogue move resulting from interpreting some segment of input:

(2) LATEST_MOVES : Oqueue([MOVE : Move; MODALITIES : Set(Modality); SCORE : Real])

In GODIS, utterance interpretations from LATEST_MOVES are added to the queue of non-integrated moves, where they stay until integrated or discarded. The representation of non-integrated moves is identical to that of LATEST_MOVES, except for the addition of a SPEAKER field (with value usr or sys):

(3) IS/PRIVATE/NIM : Oqueue([SPEAKER : Participant; MOVE : Move; MODALITIES : Set(Modality); SCORE : Real])

After integration, utterances are represented as part of the shared IS. To represent the latest and previous utterances in the SHARED part of the GODIS IS, somewhat truncated representations are used. In IS/SHARED/LU, a TURN_CONT field (representing all contents communicated in a turn/utterance) groups together all moves and corresponding scores. For IS/SHARED/PM (representing the contents of the moves in the previous utterance) only the the TURN_CONT field is retained:

(4) IS/SHARED/LU : [SPEAKER : Participant; TURN_CONT : Set([MOVE : Move; SCORE : Real]); MODALITIES : Set(Modality)]

(5) IS/SHARED/PM : Oqueue([MOVE : Move; SCORE : real])

To cater for these TIS extensions, GODIS update rules have been modified. These modifications are of limited theoretical interest and will not be discussed further here.

Utterance generation in GODIS can be regarded as a function from NEXT_MOVES : Oqueue(Move) to OUTPUT : String and OUTPUT_GUI : String. This function implements "modality fission". In the MMD approach, a very simple method of modality fission is used: generate everything both as speech and graphically, as far as possible.

The TIS variable OUTPUT : String holds the text to synthesise, and TIS variable OUTPUT_GUI : String holds MMD menu constructs as described above under "DynGUI: a generic MMD interaction GUI agent".

[1]This encodes an assumption that click input is always correctly perceived by the system.

Asynchronous control for multimodal interaction

The MMD approach puts certain requirements on the dialogue system architecture. In many single-modality spoken dialogue systems, barge-in capabilities allow the user to interrupt system utterances. To allow the same freedom when the user is interacting multimodally, it is important to be able to interrupt spoken utterances using other modalities, e.g., by clicking on a button which answers a question that the system is currently asking using speech. This means that barge-in cannot be handled internally by a ASR/TTS agent, but must be handled by the dialogue manager. The system must listen for spoken as well as point-and-click input at the same time as it is speaking. This can be handled in an architecture allowing for asynchronous control, i.e., running several system components simultaneously.

TRINDIKIT4, developed in TALK, builds on TRINDIKIT3.2 but adds important features, including asynchronous control. TRINDIKIT4 is more OAA-centered than previous versions, and features a threaded control module. Apart from enabling multimodal barge-in, this also enables a degree of incrementality in input processing. Even if the system starts working on input, the user can add more information at any time. If the system was just going to say something, and the user instead continues her turn, new input will be processed before the system takes the turn.

To handle asynchronous utterance processing, the GODIS IS has been extended with two variables:

(6) INPUT_BUFFER : Queue([TOKEN : String; MODALITIES : Set(Modality); SCORE : Real])

(7) ACTIVE_INPUTS : Set(String)

The asynchronous GODIS applications uses queues for communication between modules. Active input agents monitor input continuously, and write to a single input queue INPUT_BUFFER. This variable (which has the same type as the INPUT variable) allows several input agents to function independently by writing to the same TIS variable. When the user has not provided any input in a certain amount of time, the INPUT_BUFFER contents are considered as constituting one user turn and are copied to the INPUT variable. Then, the INPUT_BUFFER is cleared and the system proceeds by parsing the input of the user turn and updating the information state.

Since INPUT_BUFFER is cleared before starting the interpretation and update phases, it is easy to keep track of any new user input. If new input arrives, the system will handle it separately as a new turn after integrating the previous input. Also, this architecture permits any number of input agents. The DME agent need not know anything about them, they operate independently on the TIS.

When the user starts speaking or clicks on the GUI, the input agent also adds an element to the TIS variable ACTIVE_INPUTS, which is a set of strings identifying input agents currently being used to provide input. The setting of ACTIVE_INPUTS to a non-empty value triggers the controller to send a message to the output agents to stop all output (barge-in). The recognised string or representation of the click is pushed to the TIS INPUT_BUFFER variable, and the element is deleted from the ACTIVE_INPUTS set.

DynGUI: a generic MMD interaction GUI agent

DynGUI, a generic GUI agent (written in Java) for graphical menu-based interaction, has been developed in TALK to enable Multimodal Menu-based Dialogue. This agent displays multimodal output in the form

Figure 2.1: DynGUI screenshots

of menu constructs, when called by the output module. When the user interacts, e.g., by clicking a button, the agent sends an OAA request transmitting multimodal input to the information state.
Two screenshots of DynGUI are shown in Figure 2.1.
The protocol for this interaction contains the following Menu Constructs (MCs)[2]:

- button(*Text*, *Input*)
  - Display button with *Text*
  - If clicked, send click(*Input*)
- menu([$MC_1, MC_2, \ldots, MC_n$]) or menu(*Text*, [$MC_1, MC_2, \ldots, MC_n$])
  - Display menu (optionally with a text label)
  - MC_i ($1 \leq i \leq n$) is a menu construct (usually buttons)
- label(*Text*): Output *Text* message
- textentrybox(Text):
  - Display *Text* and text entry box
  - When user presses "return", send *Input* were *Input* is a string of text written by the user

Constructs in this protocol are generated by multimodal GF grammars (see Section 2.2.2). This ensures that button text, labels and messages are rendered in the appropriate language. The input to generation of MMD constructs, and the output from interpretation of MMD constructs, are GoDiS dialogue moves. As an example, the GoDiS move

[2]Additional menu constructs not currently implemented in this protocol include scrollable lists. The motivation for this is that such constructs have not been necessary to implement the UGOT applications. For example, the DJ-GoDiS uses an application-specific GUI to handle scrollable lists containing playlist songs.

(8) ask({ **?action(control_playback)**, **?action(manage_playlist)** }) ("Do you want to control playback or manage the playlist")

is converted into the following corresponding multimodal output construct:

(9) menu([button('handle the player', answer(**action(control_playback)**)), button('manage the playlist', answer(**action(manage_playlist)**))])

Multimodality in application-specific GUIs

In addition to DynGUI, which handles menu-based interaction, it is often useful to have an application-specific GUI to deal with interactions which go beyond simple menus. For example, the GoTGoDiS application displays a map of tram stops where the user can click to select departure and destination stops. DynGUI input is generally unambiguous to the system. However, in application-specific GUIs, there are cases when a click can mean different things depending on the spoken input occurring in the same turn. For instance in the GoTGoDiS system (see sections 2.2.3 and 2.3.3) the clicks in the following two multimodal utterances end up having different meanings after being parsed by the multimodal GF grammar:

(10) USR> I want to go from here [clicks on chalmers] to brunnsparken

(11) USR> I want to go here [clicks on chalmers] from brunnsparken

In the first case, the click is interpreted as providing a departure stop (answer(**dept_stop(chalmers)**)), whereas in the second case it is interpreted as providing a destination stop (answer(**dest_stop(chalmers)**)). The difference in interpretation is caused by the phrases "from here" and "here", where the former leads to interpreting the click as indicating a departure stop and the latter indicating a destination stop.

Multimodal dialogue across multiple domains

In TALK, GoDiS has been extended to allow for offline plug-and-play and implicit application switching (see D2.2 [19]). The DynGUI is used by all applications implementing the MMD approach.

When running several applications simultaneously, two additional fields DOMAIN and PHRASE_ID are added to the records representing individual moves in the LATEST_MOVES queue.The DOMAIN field is set by the parser based on the subgrammar that was used to interpret the move; it is assumed that the move is intended as input for the corresponding domain. Finally, PHRASE_ID assigns a single unique number to all moves in an utterance derived from the same phrase (syntactically maximal unit) in the input string.

Multimodality in AgendaTalk

As the calendar GUI is not a menu-based interface the MMD approach for the development of AgendaTalk was not appropriate. Therefore, the AgendaTalk application does not use the MMD approach per se but still makes use of many of the features described above. Following the same approach, AgendaTalk has asynchronous control of multimodality and uses the same utterance representation with an extended Information State with ASR scores assigned to individual dialogue moves. However, utterance

generation differs as we do not want to generate speech and graphical output in parallell but instead implements advanced context-dependent modality fission. This work is described briefly in Section 2.3.4 and extensively in D3.1 [7]. Also, AGENDATALK does not make use of the DynGUI agent.

2.1.3 Multilinguality in GODIS

This section briefly describes dialogue management aspects of multilinguality in GODIS applications.

The ISU approach, as implemented in TRINDIKIT, allows modular dialogue systems where language-specific information (including multimodal grammars) is kept separate from dialogue management and it is therefore easy to preserve smooth dialogue management while switching between languages at runtime.

The core of GODIS can thus be used for different languages, graphical interfaces, and operating situations, and can therefore be easily adapted to different applications. UGOT have implemented two different methods for changing language. In DJ-GODIS, GODIS-DELUX and GOTGODIS, language is changed by clicking a checkbox in the DynGUI (see Section 2.1.2). In AGENDATALK, language switching is accomplished verbally by naming the desired language. The approach taken in AGENDATALK is the same as the approach to language switching in MIMUS, see Chapter 4.

Concerning dialogue management, no extensive modifications were needed to handle multilinguality. The TIS variable LANGUAGE keeps track of the current language used in ASR, interpretation, generation, and TTS. Using the asynchronous processing implemented in TRINDIKIT4, GODIS sets up triggers to handle language switching. When the LANGUAGE variable is set to a new value, triggers send requests to change language to the modules and agents concerned.

One advantage of using GF grammars for multimodal output is that the text components of multimodal output (e.g., text on buttons) will be generated in the appropriate language, without any additional programming effort.

2.2 GF grammar work related to GODIS applications

The tools we use for multimodal grammar development are those which have been exploited for multilingual grammar development in the Grammatical Framework (GF) developed by Aarne Ranta. GF is described in more detail by [23], in TALK deliverable D1.2a [5], and on the GF homepage:

`http://www.cs.chalmers.se/~aarne/GF/`

The GF Resource Grammar Library contains grammar rules for 11 languages, plus some more under construction. These languages are Arabic,[3] Danish, English, Finnish, French, German, Italian, Norwegian, Russian, Spanish, and Swedish. It aims to maintain an approximately equal coverage in syntax and morphology libraries for all the languages. The coverage of the resource grammar is comparable to the Core Language Engine as described in [24].

One of the UGOT tasks in TALK has been the integration of GF with the dialogue system development toolkit TRINDIKIT.[4] This has involved the development of a library of multilingual and multimodal grammars related to some of the showcases involved in TALK. These grammars have related abstract syntax to

[3] The Arabic grammar does not cover the full resource API yet.

[4] `http://www.ling.gu.se/projekt/trindi/trindikit/`

several different concrete syntaxes corresponding to different natural languages and non-linguistic modalities. Another UGOT task has involved extending these grammars to integrate several modalities, such as speech and pointing gestures. This has essentially consisted of extending existing concrete syntaxes with multimodal information.

One of the main reasons to use GF in connection with GODIS/TRINDIKIT is the possibilities for rapid prototyping and fine-tuning of dialogue systems. GODIS/TRINDIKIT is well suited for rapid development of dialogue systems, and to this we now can add rapid development of grammatically correct, multilingual and multimodal grammars for GODIS dialogue systems.

To connect the application grammars with the GODIS dialogue manager, we have developed a multilingual and multimodal resource grammar which contains application-independent utterances, pointing gestures and combinations thereof. This common resource grammar reduces the sizes of the application specific grammars by 25%–75% in our showcase systems.

The common GODIS resource grammar is in turn implemented by using the GF Resource Grammar Library. Of the 11 languages existing for the Resource Grammar Library, we have implemented 7 in the GODIS resource grammar. These languages are English, Finnish, French, German, Italian, Spanish, and Swedish. One of the showcase systems, the GOTGODIS Tram information system, described in section 2.2.3, is multilingual in all 7 languages.

Multilinguality

The separation of abstract and concrete syntax makes GF well suited for developing multilingual grammars. Each language becomes one concrete syntax of a common abstract syntax. Also, it is possible to write language-independent grammars, where the grammar rules are represented as syntax trees of a multilingual resource grammar.

The common GODIS resource grammar is designed with the purpose of making it easy to write multilingual application grammars. This means that the only thing that has to be provided when adding a new language is a lexicon of a few application-specific lexemes. For our example applications, four out of five are written as language-independent grammars.

Multilinguality is described in more detail in deliverables D1.1 [16] and D1.5 [14].

Multimodality

The possibility of several concrete syntaxes also opens up the possibility for multimodal grammars, where each modality is a separate concrete syntax. For the GODIS applications we have implemented two kinds of multimodal utterances, parallel and integrated multimodality (see deliverable D1.2a [5] for definitions).

Parallel multimodality is when an utterance can be presented in either modality separately. In the GODIS resource grammar there is support for multimodal system utterances, both via text-to-speech and through a GUI. This is made solely in the resource grammar, which means that the application grammar does not have to contain any multilingual utterances.

Integrated multimodality is when the meaning of an utterance can only be deduced from looking at several modalities simultaneously. This is implemented in the GODIS resource grammar by making it easy for the application grammar writer to specify when e.g., a click in the GUI can be combined with an utterance.

Multimodal integration is described in more detail in deliverables D1.2b [15] and D1.5 [14].

Application switching

Since there is a common resource grammar for GODIS applications, it is very straightforward to combine the grammars for two (or more) different applications into one unified grammar. One only has to write a wrapper grammar importing the different application grammars.

In GODIS there is always one *active* application which listens to what the user says. Sometimes one would not want to recognize all possible utterances for the inactive applications, but only e.g., the main requests and questions. This is solved by splitting each application grammar into one global and one local grammar, where the global grammar contains all utterances which should be recognized whichever application is active, and the local grammar contains utterances which only can be uttered when the application is active.

Application switching is described in more detail in deliverable D2.2 [19].

Compilation to low-level formats

From any GF grammar it is possible to create grammars in other formalisms, e.g., context-free grammars. The possible context-free output formats include Nuance Grammar Specification Language (GSL), JavaSpeech Grammar Format (JSGF), and the W3C Speech Recognition Grammar Specification (SRGS). This means that the same GF grammar can be used both for speech recognition and for parsing the recognized utterance. This is described in more detail in deliverable D1.1 [16].

Another possibility is to create a corpus of utterances from a GF grammar, which then can be used for training an SLM, which also can be used in speech recognition. This is described in more detail in deliverable D1.3 [30].

Structure of this section

The rest of this section contains short descriptions of the grammars for our example applications. First a non-GODIS application is presented. Then there is a description of the common grammar library for writing grammars for GODIS applications. Finally there are descriptions of the grammars for four GODIS applications: GOTGODIS, AGENDATALK, DJ-GODIS, and GODIS-DELUX.

2.2.1 Grammars for GOTTIS

This section briefly describes a multimodal and multilingual GF grammar that has been developed for the GOT Tram Information System (GOTTIS). GOTTIS was one of the first example systems developed in the TALK Project, and is described in more detail in deliverable D1.2a [5].

GOTTIS is not a GODIS application, and thus has a very limited dialogue manager. We still include a description in this chapter, as an example of a "baseline" system with respect to dialogue handling capabilities. The system has been rewritten into a GODIS application called GOTGODIS, which is described later in section 2.2.3.

Multimodal input

User input is done with integrated speech and pointing modalities. The user may use speech only, or speech combined with pointing gestures on the map. Pointing gestures are expected when the user makes

a query containing “here” (though “here” might also be used without gestures, see below). The supported pointing gestures are clicks on stops, and drawings around a set of stops.

A pointing gesture is represented as a list of stops that the gesture refers to. For a click, this is normally a list containing a single bus/tram stop, but some stops might be close enough that a click could refer to more than one stop. The set might also be empty if the click was not close to any stop. For map drawings, the list of stops contains the set of stops in the area drawn on the map.

In the concrete syntax, the pointing data is appended to the speech input to give the parser a single string to parse. These are some examples using the English concrete syntax:

- “i want to go from brunnsparken to vasaplatsen;”
- “i want to go from vasaplatsen to here; [Chalmers]”
- “i want to go from here to here; [Chalmers] [Saltholmen]”

Indexicality

To refer to her current location, the user can use “here” without a pointing gesture, or omit either origin or destination. The system is assumed to know where the user is located. Examples in English concrete syntax:

- “i want to go from here to centralstationen;”
- “i want to go to valand;”
- “i want to come from brunnsparken;”

Ambiguity

Some strings may be parsed in more than one way. Since “here” may be used with or without a gesture, input with two occurrences of “here” and only one gesture is ambiguous:

- “I want to go from here to here; [Valand]”

A query might also be ambiguous even if it can be parsed unambiguously, since one click can correspond to multiple stops:

- “I want go go from Chalmers to here; [Klareberg,Tagene]”

The current application fails to produce any output for ambiguous queries. A real system should handle this through dialogue management.

Multimodal output

The system’s answers to the user’s queries are presented with speech and drawings on the map. This is an example of parallel multimodality as the speech and the map drawings are independent.

The information presented in the two modalities is however not identical as the spoken output only contains information about when to change trams/buses. The map output shows the entire path, including intermediate stops.

Parallel multimodality is from the system's point of view just a form of multilinguality. The abstract syntax representation of the system's answers has one concrete syntax for the drawing modality, and one for each natural language. The only difference between the natural language syntaxes and the drawing one is that the latter is a formal language rather than a natural one.

Multilinguality

Currently, speech input and output in English and Swedish are implemented. The dialogue system itself accepts input in either language, but speech recognizers can often only handle a single language at a time.

System output is linearized using the same language as the speech input was in.

Adding support for a new language requires writing concrete syntaxes for the user and system grammars.

2.2.2 Grammars common to all GoDiS applications

We have created a unified GF grammar library covering all our GoDiS applications – GoTGoDiS, AgendaTalk, DJ-GoDiS, and GoDiS-deLux. Much work has been spent on the design of the library, making it simple to write a grammar for a new domain, a new language, or a new modality.

The reason why the GOT Tram Information System (GOTTIS) is not included in the library is that GOTTIS is not based on the GoDiS dialogue manager.

The grammar library is split into two main parts – system utterances and user utterances. The dialogue system has many different ways of talking about issues, predicates, actions, feedback, etc. e.g., a predicate can be mentioned in a wh-question, y/n-question, an answer, a report, or several different feedback moves. Thus the grammar for system utterances must be a grammar with a wide linguistic coverage, which produces grammatically correct utterances for different dialogue moves.

This would be useful for the user grammar too, if it hadn't been for speech recognition. The main focus of the user grammar is to restrict the number of utterances to only the ones that are really used, to be able to improve speech recognition. Also, in some application domains there are commonly used utterances that are not grammatically correct sentences, but should be recognized anyway. Thus the user and system grammars are quite different in their design, but there are still things that can be shared between the grammars.

System utterances

An utterance in GoDiS consists of a sequence of dialogue moves. For the system grammar, each dialogue move is a separate utterance, which is mostly realized as a sentence, but other forms are possible, e.g., noun phrases.

Entities in GoDiS such as actions, predicates, propositions, and questions, all have their counterpart in the grammar. For each way of building an entity, such as building a proposition from a predicate and an individual, or building a y/n-question from a proposition, there is one or several corresponding grammar rules. e.g., there is a rule for applying a predicate to an individual (forming a proposition), and there is a rule for turning a proposition into a y/n-question.

Many GODIS entities can be used in several different ways by the dialogue system. e.g., an action can be uttered as a request ("play a song"), as a y/n-question ("do you want to play a song?"), as a report ("I am playing a song"), or in feedback and grounding moves ("you want to play a song, is that correct?"). In some languages all these different phrases can have different realizations, which means that it is not possible to store the action as a pure string. Instead we make use of the GF Resource Grammar Library to define the realizations for each GODIS entity.

Here is a small list of the main GODIS entities, and the corresponding grammatical categories in the GF Resource Grammar Library:

Dialogue moves are realized as sentential phrases (which can consist of sentences, questions, noun phrases, etc.)

Actions are realized as verb phrases

Propositions are realized as declarative clauses including all tenses

Questions are realized as question clauses including all tenses

Short answers are realized as noun phrases

How to form a realization of a dialogue move from e.g., an action depends on many variables – things such as politeness level (e.g., should we use the polite "you" or the second person "you" in German or Swedish), which exact words to use (e.g., should we use "I want to" or "I would like to"), and how informative we want to be (e.g., should we say "what" or "I'm sorry, I didn't hear what you said"), play a role and are often different for different languages. Therefore the specific grammar rules can be fine-tuned for different languages.

The fact that the common system grammars are specialized for then different languages also has the effect that application-specific details can be made very language-independent. Thus, it is often simple to implement the application grammar as a language-independent grammar.

Multimodal system utterances

We have extended the system grammar to automatically generate utterances in a tailor-made GUI description language. Different kinds of dialogue moves are realized as different GUI elements. As an example, an alternative question is realized as a menu of choices, whereas a wh-question is realized as a text entry box.

The multimodal grammar reuses the previously described utterance grammar for producing the text inside buttons, menus and labels. This means that multimodal system output comes for free when designing a new application grammar, in all languages for the application domain.

User utterances

The common grammar for user utterances has a simpler structure. There are only four kinds of user utterances – questions, actions, answers and short answers. On the other hand, each utterance can correspond to several dialogue moves in sequence. A user utterance can also contain partial answers to some of the follow-up questions. An example is the utterance "turn on the bedroom light", which is categorized as a

user request, but which also gives partial information about which light to turn on. One part of the utterance ("turn on the ... light") corresponds to a GoDiS request move, while another part ("the bedroom") corresponds to a GoDiS short answer.

The application grammar then has to list the possible user utterances, and connect them to a sequence of dialogue moves. In many cases the system grammar can be used as a resource for specifying either the dialogue moves or the utterances.

It is not necessary to specify the user grammar as a language-independent grammar. However, by doing this it becomes very simple to add a new language for the application domain. The only thing that has to be done is to translate an application-specific lexicon, which in our example domains consists of not more than 30–50 lexicon entries.

Multimodal user utterances

Multimodality is added to user utterances by adding a new constituent to the linearization categories of questions, actions, answers and short answers. The new constituent holds the pointing gestures and clicks the user makes during the input phase.

Several useful operations are defined for making it possible to specify that some utterances (e.g., "play this song") should have an associated click, while other utterances (e.g., "play a song") should not be uttered in association with a click.

2.2.3 Grammars for GoTGoDiS

Implemented languages

All grammars for the GoTGoDiS domain are written as language-independent grammars, where the utterances are specified using syntax trees from the GF Resource Grammar Library. For the language-specific lexicon entries there is a lexicon grammar with about 20 entries.

The GoTGoDiS domain is implemented in 7 different languages, of which one (Finnish) is not an Indo-European language. The time it takes to implement a new language for this application is less than one day, for a fluent-speaking person who is a fairly competent GF grammar writer.

Tram stops and tram lines

The stops and lines of the trams, busses and ferries of the Gothenburg public transport system, are stored as a multilingual grammar resource which is used by both the system and the user grammar.

Each line and stop has a corresponding grammar rule, which is linearized in different ways in different languages. Especially for stops this can be very language-dependent – some places can have natural translations, whereas others should be kept (e.g., proper names).

The system grammar

There are only three predicates and two actions in GoTGoDiS. Of the predicates one can be asked for by the user ("what is the shortest route?") and two by the system ("where do you want to go to?" and "where do you want to go from?"). The actions are general actions for restarting the system, and requesting help.

When the system answers the question about the shortest route, it answers with a list of lines and departure/destination stops. The system utters this list as a grammatically correct sentence, while simultaneously drawing the route on the map.

The user grammar

The user grammar consists of different ways the user has of specifying departure and destination stops, by clicking or speaking.

The grammar recognizes simple departures and destinations, such as “from Chalmers” or “to Klippan”, but also combinations of these, such as “from Chalmers to Klippan”. Clicks in the map, and combinations of clicks and utterances, are also recognized by the same grammar. e.g., “from Chalmers to here” together with clicking in the map is a valid utterance.

Furthermore, the grammar recognizes underspecified utterances, i.e., uttering only a location (“Chalmers”), and combinations of underspecified locations and departure/destinations, such as “Klippan from Chalmers”. As before, clicks and combinations of clicks and utterances, are recognized by the grammar, e.g., “Klippan from here” together with clicking in the map.

There are several alternative ways of saying the same thing, both by combining clicks and utterances in different ways, and by saying things like “I want to go to Klippan” or “to Klippan, please”.

2.2.4 Grammars for AGENDATALK

The goal for the AGENDATALK GF grammars was to generate a corpus for SLM creation for user utterances without standing in the way for more extensive use in the future. Thus the grammar is based on the common GODIS resource grammar, making use of the unimodal structures and general functions. As focus is on the creation of an SLM, there is no extensive GF system grammar for in use AGENDATALK. System output is instead handled by a GODIS Prolog resource.

Implemented languages

The foundation of the AGENDATALK grammar is a minimal language independent structure built using the resource grammar. It is then implemented in two languages; English and Swedish.

The Booking resource

The Booking resource is a domain specific collection of items that are used in order to compile the AGENDATALK specific concept of a booking. It contains the classes Time, Date and Event.

Event consists of actual events such as *meeting* or *lunch*

Time consists of time related items like *at four*, *nine o’clock*, *in the morning*

Date consists of date related information including *December fifteenth*, *Monday the sixth*

A booking, in other words, is a composition of one or more of those classes, for example the following which is made up of an Event, a Time and a Date: *”meeting at four on the fifth of December”*.

In the future, other classes such as Location and Person could be incorporated, allowing for bookings such as the following, which makes use of Event, Person, Location, Date and Time: *"lunch with Peter at Plaza tomorrow at noon"*.

The user grammar

As opposed to the other grammars presented here, the AGENDATALK grammar is not used for parsing at runtime and the goal is to focus on coverage. The user utterances are written to be as extensive as possible to form a large enough corpus for SLM generation. The grammar as a whole covers ten short answers and their combinations, five predicates and six actions and their different combinations with answers. In its current state, disregarding negations, this means a little over 4 million utterances for the English corpus and just under 2 million for the Swedish.

Because of the structure of the AGENDATALK system as a whole the user utterances are considered unimodal. As the common resource library is adapted to allow for multimodal grammar building it is necessary to explicitly call a function which marks the different modalities of a certain type of utterance. In the case of AGENDATALK they are all to be considered unimodal, and has been marked as such by using the noClick function supplied by the resource library. See code example below where we show a simple delete request without any embedded answers to be recognized by GODIS.

```
delete =
   noClick (variants{ req1  delete_variants;
                      reqVP agenda_delete;
                      req1x delete_variants
                            (optStr(a_booking_variants ++ from_agenda))});
```

This code shows the function "delete" which defines the different ways the user can prompt the system to initiate the delete plan without also supplying answers. As mentioned before we have defined the delete function as unimodal using the noClick function. The utterances are then linearized in several different variants. Apart from agenda_delete, which is defined using the AGENDATALK lexicon, we have a list of possible synonyms to initiate the delete action, i.e., *delete_variants*. The delete_variants is a list of verbs such as *remove* or *erase*. They are all specified with additional functions: req1, reqVP and req1x which are functions from the common grammar resource's system independent framework. This adds, among other things, to the utterance structure including the initial *"I would like to"* and the closing *"please"*.

These can then be followed by things that have no semantical significance to the system, *"something"*, *"a booking"*, *"to the agenda"* or *"to my calendar"* for instance. These differ from the structural additions inherited from the resource library in the sense that they are application specific.

The resulting user utterances in English for a simple delete command then include:

```
"delete"
"remove an event please"
"i would like to delete something from my agenda"
```

Diverting from the structure Earlier we presented the general structure of the resource grammar. Among other things we defined a GODIS question as question clauses. In the AGENDATALK User grammar there are several GODIS questions that are not at all posed as questions by the user. For example

"show me the schedule", which is an ask(X^schedule(X)). These are easily added to the user grammar without running into problems with the underlying structure.

Pronoun resolution The AGENDATALK system has a solution for pronoun resolution. The user has the possibility of implicitly referring to the object in discussion. If, for example, it has been established that user and system are talking about a meeting the user can simply prompt the system to *"add it"* and it will be interpreted as "add meeting". This type of resolution is easily handled in the GF grammar.

An interesting side-effect of this grants us the possibility to let the user say things like "put it back", which will indicate the booking that was just manipulated.

2.2.5 Grammars for DJ-GODIS

Implemented languages

The grammars for the DJ-GODIS mp3 domain have a language-independent backbone implemented using the GF Resource Grammar Library. Two languages are then implemented as extensions of the obtained syntax trees, Swedish and English. This is done by using a language-specific lexicon of roughly 40 words.

Musical resources

The musical resource for the mp3 domain consists of two parts, one for artists and one for songs. The songs are defined as Song and the artists as Artist. So "Lucky star" is a Song and "Madonna" is an Artist. Song and Artist are then used when defining the functions in the system and user grammars. Both songs and artists are linearized using the GF Resource Grammar Library's NP type in order to easily fit them into the concrete syntax.

System utterances

The system grammar consists of 15 predicates, three short answers and 16 actions. Four of the predicates are for grounding issues when the user has clicked in the DJ-GODIS GUI. The other predicates deal with questions about adding, deleting and playing songs and artists. Furthermore there are predicates for what the current song is and which songs or artists are available.

Short answers can be a song, an artist or a song and an artist. "Abba", "Waterloo" and "Waterloo by Abba" are all possible short answers.

Of the actions two are the general GODIS actions for returning to the top menu and giving the user help. Then there are two actions representing the next level in the menu structure; controlling playback and managing the playlist. Controlling playback can be done by playing, pausing, fast forwarding/rewinding or controlling the volume (which in turn can be done by lowering or raising the volume). Managing the playlist can be done by adding or deleting an item or clearing/shuffling the playlist.

User utterances

The user side of the grammars has predicates for asking about available songs and artists and the name of the song currently being played. There are also two predicates for when the user clicks in the DJ-GODIS GUI.

There are three ways of giving non-spoken input in the user grammar: The first alternative is by clicking the DJ-GODIS GUI, the second alternative is to click the generic DynGUI and the third alternative is to write in DynGUI's text field which corresponds to giving spoken input through the TTS agent.

Combining the two modalities gives three possible modality configurations:

- Speech only: "I want to play a song by madonna"
- Graphics only: "click(answer(artist_to_play(madonna)))"
- Speech and graphics: "Can you play this artist ; click(answer(artist_to_play(madonna)))"

User utterances for multiple applications

Since the DJ-GODIS mp3 application is used together with the GODIS-DELUX application the user utterances are split into two grammars, a global grammar called MP3Global and a local called MP3User. MP3User is an extension of MP3Global.

The division used for the MP3 user grammars is done by restricting the help and top questions to MP3User together with the predicates for available artists and songs. This means that the currently active application has to be the MP3 player in order to ask for help or get an answer to what songs there are by Madonna. Also, all short answers are declared in the local grammar since they need the context of the mp3 domain to be meaningful.

MP3Global has all functions that can be accessed while other applications are in focus. That means that the global grammar holds all other actions together with the question for finding out the name of the current song. Furthermore MP3Global is where the click-only-predicates for playing or deleting a song in the playlist and playing or adding a song in the media library can be found.

MP3Delux is then obtained by combining the MP3User grammar with the DeluxGlobal grammar. For English all that is needed are the following two lines:

```
--# -path=.:../DeLux:../Common:prelude:alltenses
concrete MP3DeluxEng of MP3Delux = MP3UserEng, DeluxGlobalEng ** {}
```

The resulting grammar covers all possible user actions, short answers and predicates for the mp3 domain and all predicates and actions (except the requests for the top menu and help) for the GODIS-DELUX domain.

2.2.6 Grammars for GODIS-DELUX

Implemented languages

All grammars for the GODIS-DELUX domain are written as language-independent grammars, where the utterances are specified using syntax trees from the GF Resource Grammar Library. For the language-specific lexicon entries there is a lexicon grammar with about 20 entries.

This domain is implemented in 2 different languages, Swedish and English. Since the language-dependent parts of the grammars are about the same size as the GOTGODIS grammar in section 2.2.3, the time it will take to implement a new language for this application should be less than one day, for a fluent-speaking person who is a fairly competent GF grammar writer.

Lamps and rooms

The rooms and lamps in the GODIS-DELUX domain are stored as a multilingual grammar resource which is used by both the system and the user grammar.

The rooms grammar contains the different rooms in the example house, e.g., “the bedroom” and “the kitchen” The are even two grammars for lamps – one with underspecified, general types of lamps which the user can say, e.g., “the floor lamp” and “the ceiling lamp”. There is also a grammar containing the specific lamps in the example house, e.g., “the bedroom floor lamp” and “the kitchen ceiling lamp”.

Each room and lamp has a corresponding grammar rule, which is linearized in different ways in different languages.

The system grammar

There are five actions and four predicates. Of the actions, one is a general action for returning to the top menu. The other four actions are about turning on and off, and dimming and undimming, lights. The four predicates are used when the user asks which lamps are on/off, or if a specific lamp is on/off.

There are also a couple of auxiliary predicates which are used for requesting information about which lamp a given action is supposed to act on.

The user grammar

The user grammar consists of different ways the user has of either requesting things to be done with lamps, or asking about the state of lamps. This can be done either by speaking, or by modifying the lamps directly (i.e., turning on/off physically).

The grammar recognizes action requests, both simple (e.g., “turn on the light”) and combined with additional information (e.g., “turn on the floor lamp in the kitchen”). There are only minor differences in how the four modification actions are uttered (e.g., “turn on” vs “turn off”), and therefore we have unified all four actions into one single grammar rule taking an extra argument specifying the action to use. The grammar also permits the user to specify simple quantification, namely to mention *all* lamps or *all* rooms in an action request.

The user can ask questions about the state of lamps, i.e., which lamps that are on/off, or if a specific lamp is on/off. This can be done as a simple utterance (e.g., “which lamps are on”), or combined with additional information (e.g., “which lamps are on in the kitchen”). Since there is only a minor difference between asking whether a lamp is on or off, the on/off questions have been combined into one single grammar rule.

Furthermore, the grammar recognizes underspecified utterances, i.e., uttering only a room (“the bedroom”).

User input for multiple applications

The GODIS-DELUX domain is intended to be used together with other applications. Some of the domain-specific user utterances are only supposed to be recognized when the GODIS-DELUX application is active, e.g., short answers consisting of only a room or a lamp should not be recognized when another application is active.

This is solved by splitting the user grammar into two grammars, one global and one local grammar, as described in more detail for the DJ-GODIS application in section 2.2.5. The global grammar contains all

utterances which should be recognized whichever application is active. The local grammar extends the global utterances with e.g., short answers. Finally there is a wrapper grammar which imports the local GODIS-DELUX grammar and the global grammars from the other applications, and this becomes the multiple-application GODIS-DELUX grammar.

2.3 GODIS applications demonstrating multimodality and multilinguality

2.3.1 Introduction

This section describes the GOTTIS and GODIS in-home applications demonstrating multimodality and multilinguality. We also provide examples of running GODIS with multiple domains. While all applications are useful in an in-home environment, several of them would also be useful in other environments.

For each application, we describe a scenario, the application infrastructure, the research issues addressed, application functionality, multimodal and multilingual aspects of the application, and the application resource implementation. We also provide transcripts of example interactions.

2.3.2 GOTTIS

GOTTIS is a multimodal demonstration system for finding the shortest route through the Gothenburg public transit system. GOTTIS is not a GODIS application, but its purpose is to clearly demonstrate the grammar-based approach to multimodality. GOTTIS forms the basis for GOTGODIS and is described in more detail in deliverable D1.2a [5].

Scenario

For example, a user might be at Brunnsparken, and wants to know which is the shortest route to Lindholmen. She tells the system that she wants to go from Brunnsparken to Lindholmen. She can give this information to the system using speech only, or a combination of speech and clicks on the map. The system responds with the route she should take. The route is shown on the map and spoken by the system.

Infrastructure

GOTTIS uses the Embedded GF Agent, NuanceWrapper and the Map Agent. The grammars used by the Embedded GF Agent are produced by the GF system.

Research issues addressed

This system is the main demonstrator for the grammar-based approach to multimodality described in D1.2a.

Functionality

GOTTIS can find and present the shortest route through a public transit network, given information about the origin and destination stops, and a database describing the network. The system has a concept of a fixed current location, which is assumed to be the location of the user when indexical expressions are used.

Multilinguality

The system supports English and Swedish input and output. This is achieved by having multilingual GF grammars for user and system utterances. Nuance GSL grammars are generated automatically from the user grammars.

Multimodality

User input is done with integrated speech and pointing modalities. The user may use speech only, or speech combined with pointing gestures on the map. Pointing gestures are expected when the user makes a query containing "here" (though "here" might also be used without gestures, see below). The supported pointing gestures are clicks on stops, and drawings around a set of stops.

The system's answers to the user's queries are presented with speech and drawings on the map. This is an example of parallel multimodality as the speech and the map drawings are independent. The information presented in the two modalities is however not identical as the spoken output only contains information about when to change trams/buses. The map output shows the entire path, including intermediate stops.

The map is a generic OAA agent for displaying graphs with positioned nodes and a background image. Other agents can highlight paths in the graph and query the map agent for node selections made by the user.

Implementation of application specific resources

The transit network information represents a subset of the Gothenburg transit network. The shortest path between two stops is calculated use Dijkstra's shortest path algorithm. The database only includes stops and time between stops, not departure and arrival times.

Dialogue examples

English The user says "i want to go from chalmers to here" and clicks on Frihamnen.

The system produces the speech output "Take 6 from Chalmers to Vasaplatsen. Take 2 from Vasaplatsen to Brunnsparken. Take 5 from Brunnsparken to Frihamnen." It also draws a path on the map, as shown in figure 2.2 on page 25.

Swedish The user says "jag vill åka från chalmers hit" and clicks on Frihamnen.

The system produces the speech output "Ta 6 från Chalmers till Vasaplatsen. Ta 2 från Vasaplatsen till Brunnsparken. Ta 5 från Brunnsparken till Frihamnen." (The English translation for this example corresponds to the English example given just above). The map looks as in the previous example.

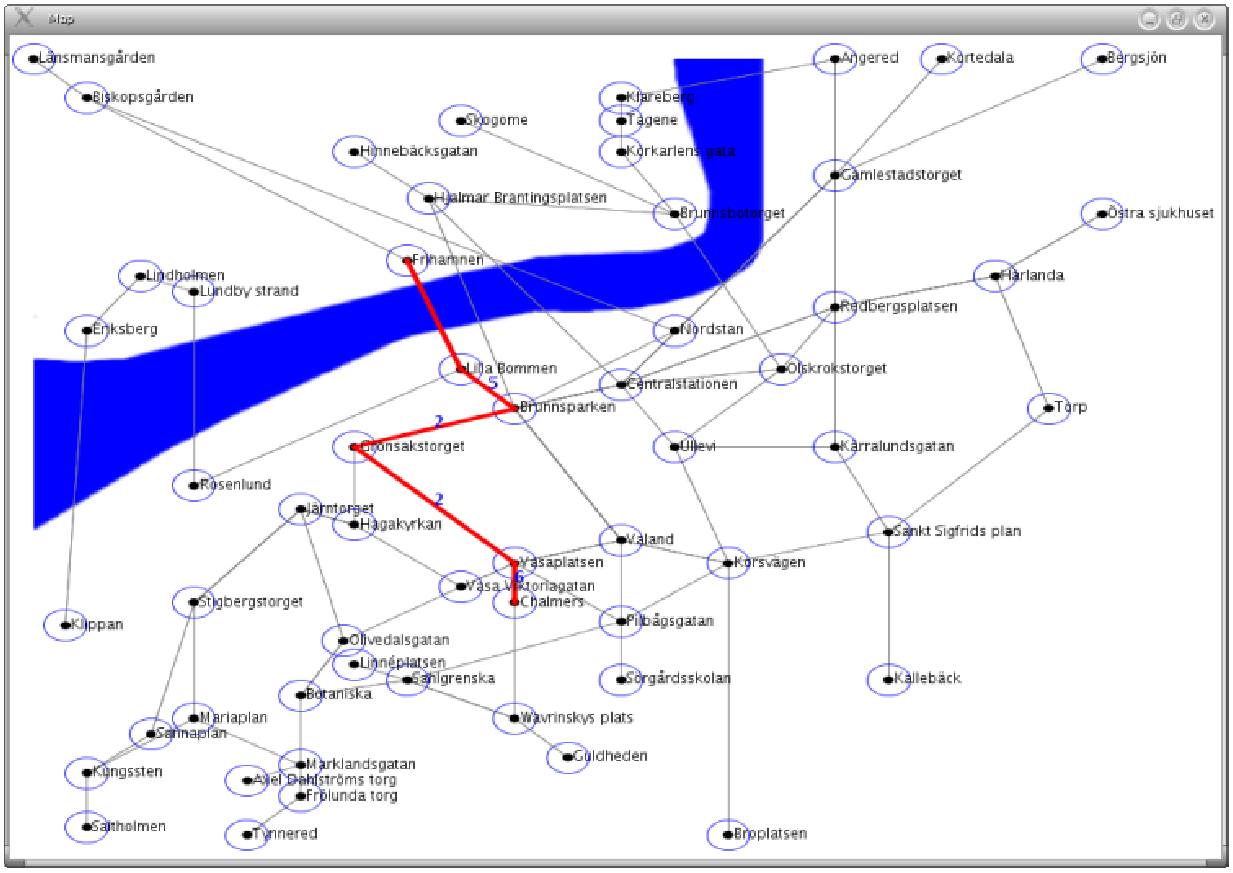

Figure 2.2: The map showing the path from Chalmers to Frihamnen.

2.3.3 GoTGoDiS

GoTGoDiS (Gothenburg Tram GoDiS application) is a multimodal, multilingual route planning system for the Göteborg tram/bus network for public transportation. Using speech and map clicks the user can supply a departure and a destination stop and the system answers with a description of the shortest route to take between the two stops. GoTGoDiS is based on the GOTTIS system described in D1.2a [5], which does not use the GoDiS dialogue manager, .

Scenario

In an in-home setting, the GoTGoDiS application can be used to quickly get tram route information using multimodal interaction. However, for a slightly more exciting scenario (requiring some additional functionality compared to the current prototype[5]) we can imagine the application being available in a public place. This is also a setting where multilinguality is extremely useful.

(12) *The application is located in the Gothenburg bus terminal with a touch screen interface. An English tourist wants to find the best way to travel with public transportation from the terminal to his hotel. The screen reads "Hej, Vad kan jag göra för dig?" (Eng. "Hello, What can I do for you?"*

U: [user presses a button to select English as language]

S: What can I do for you?

U: I need to go to Quality Hotel 11

System assumes that the departure stop is the bus terminal

S: Okay. What time do you want to leave?

U: As soon as possible

S: Take Bus 17 at 13.21 from Centralstationen to Nordstan, then take Bus 16 from Nordstan to Eriksberg.

S: Would you like to download this information to your mobile phone?

Being new in town the user feels it is a good idea

U: Yes

Graphically indicates that downloading has begun

S: Download finished

A route planning application, enabling interaction in, say, 10 or 15 different languages, could be very useful for helping people, in their own native language, to get around using the local public transportation network.

Infrastructure

The application uses the GoDiS dialogue manager and the Trindikit4 dialogue system toolkit and consists of a collective of OAA agents organized as in figure 2.3. The Controller agent, DME agent, and the

[5]Specifically, adding timetable information and the ability for the user to download relevant information to her mobile phone.

MMD, ASR, and TTS agents are TRINDIKIT agents, which communicate using the Trindikit4 OAA API described in deliverable D1.2a [5].

- The controller agent coordinates the different modules and agents by executing a set of serial control algorithms in parallel.
- The timeout agent is used by the controller to determine when the user's turn is over.
- The DME agent holds the total information state (TIS) and the core dialogue management modules, update and select, as well as interpretation and generation modules.
- The actual interpretation and generation is done by the GF agent, which is called over OAA by the interpretation and generation modules.
- The DynGUI input/output module agent is used to dynamically render graphical menus which can be used for graphical input.
- The ASR module agent continuously listens for input and writes the recognized result to TIS.
- The TTS module agent reads output from TIS and synthesizes it as speech, when called from the controller.
- The Map agent graphically draws the route description on the map and also offers a possibility for the user to provide graphical input by clicking on the tram and bus stops displayed on it.
- The Graph agent is used to compute the shortest route to take between two stops.

Research issues addressed

The GOTGODIS application addresses the following research issues:

Multilinguality and Rapid Application Localisation Using GF resource grammars, GOTGODIS has been localised to 6 languages in addition to English: Swedish, German, Spanish, French, Italian, and Finnish.[6] As mentioned in Section 2.2.3, localisation of the GOTGODIS prototype application to a new language takes about a day for a fluent speaker of the language who is also a fairly competent GF grammar writer. Related work is reported in D1.1 [16].

IBDM and Rapid application prototyping The GOTGODIS application is implemented using a single dialogue plan. Nevertheless, because of the domain-independent theory of Issue-Based Dialogue Management (IBDM) implemented in the GODIS DME (see also D5.1.2 [3], D2.2 [19]), a wide range of advanced dialogue management features are available in the application.

Integrated approach to multilinguality and multimodality By deploying multimodal (and multilingual) GF grammars, GOTGODIS demonstrates the integrated approach to multimodality and multilinguality described in Section 2.2 as well as in D1.2b [15] and D1.5 [14].

[6] For some of these languages, ASR and TTS are not readily available; instead, text input and output facilities are provided.

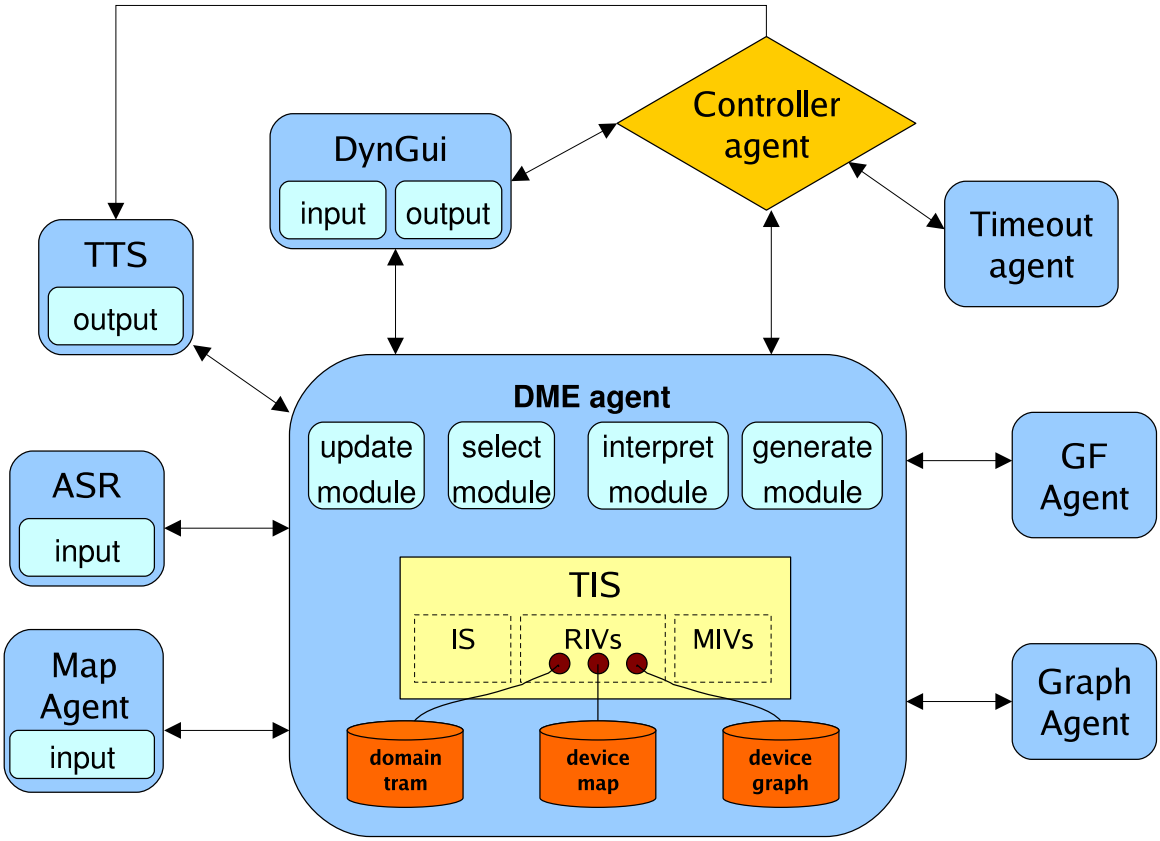

Figure 2.3: GoTGoDiS system as a collective of agents

Functionality

The only functionality of GoTGoDiS is to answer user questions about the shortest route between two stops. In the GOTTIS system, the user could do this by combining speech and map clicks, e.g.:

- I want to go from brunnsparken to vasaplatsen
- I want to go from vasaplatsen to here [user clicks on brunnsparken]
- I want to go from here to here [user clicks on brunnsparken and vasaplatsen]

As mentioned above, GOTTIS has no real dialogue manager, which results in problems when interacting with GOTTIS using speech. For example:

- It requires that users supply all necessary information in a single utterance (in this case the issue to be solved, a destination stop and a departure stop).
- It requires that an utterance has exactly one interpretation. Ambiguous user utterances cannot be handled. For example, the system cannot ask for clarification if faced with an utterance like "chalmers, find the route" - does the user mean from chalmers or to chalmers?
- It cannot deal adequately with ASR errors. The user utterance must be correctly interpreted the first time.

With GoTGoDiS we take advantage of the domain independent dialogue management that GoDiS supplies. In Section 2.3.3 we look at example interactions with GoTGoDiS to show how dialogue management has increased both the usability and robustness of the application.

Multilinguality

GoTGoDiS now supports interaction in seven languages; English, Swedish, Finnish, German, Spanish, French and Italian. GF grammars are used for both parsing and generation.The GF grammars for GoTGoDiS are described in Section 2.2.3.

Multimodality

The available user input modalities are speech, DynGUI manipulation and map clicks. Speech and GUI interaction may be combined in a single utterance. The system output modalities are speech and map drawings.

System output about a route is presented by both spoken output, DynGUI output and by drawings on the map. The map is a a schematic map of the public transportation network in Gothenburg. The OAA Map agent is described in D5.1.2 [3] and is basically the same as the one used in the GOTTIS application

Figure 2.4 shows a screen-shot of the map with drawing instructions indicating the shortest route between Olivedalsgatan and Torp.

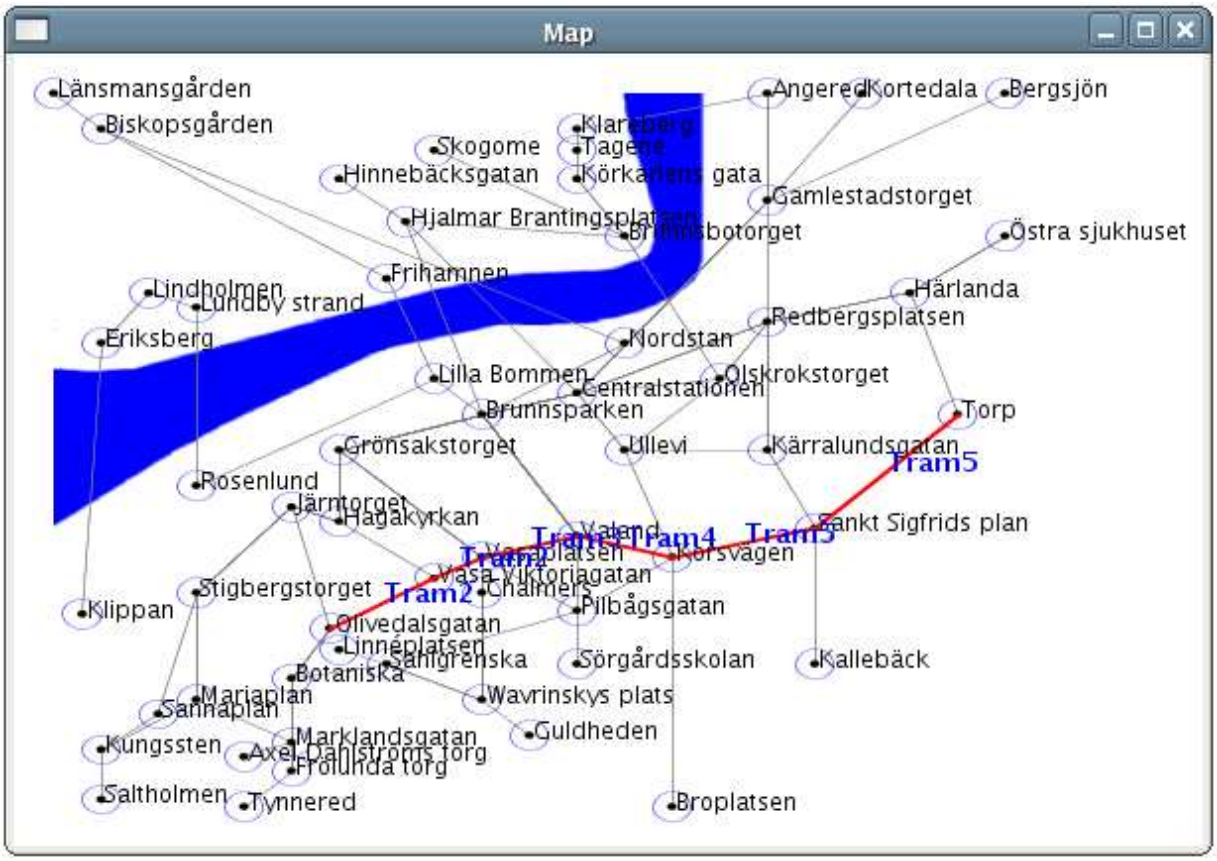

Figure 2.4: Output map drawings after the user input “I want to go from here [clicks on olivedalsgatan] to Torp”.

Implementation of resources

Domain resource In GoTGoDiS, there are only two dialogue plans which govern dialogue management; the **top** plan (which prompts the user with an open question, e.g., "What can I do for you?") and the **?*x*.shortest_path(*x*)** plan.[7]

The plan for dealing with the issue **?*x*.shortest_path(*x*)** is shown below:

ISSUE : **?*x*.shortest_path(*x*)**
PLAN:
- findout(**?*x*.dept_stop(*x*)**)
- findout(**?*x*.dest_stop(*x*)**)
- dev_query(graph,**?*x*.shortest_path_expl(*x*)**)
- dev_query(graph,**?*x*.shortest_path(*x*)**)
- dev_do(map,DrawRoute)

The two findout plan constructs first in the plan are used to get required information about a departure stop and destination stop. The device query dev_query(graph, **?*x*.shortest_path_expl(*x*)**) then queries the graph device for the shortest route between the specified departure and destination stop. The returned representation of the route consists of a list of legs on the following format:

```
leg(Line, Dept_stop, Dest_stop, Weight)
```

The representation returned from the Graph agent might look like this:

```
[leg('5', 'torp', 'harlanda', '3.0'), leg('5', 'harlanda',
'redbergsplatsen', '2.0'),leg('5', 'redbergsplatsen',
'olskrokstorget', '2.0'),leg('5', 'olskrokstorget',
'centralstationen', '3.0')]
```

This explicit representation is needed for the Map agent to draw the route on the map, but for the internal representation in GoDiS we can omit legs which do not require changing to a different tram line. This motivates the second device query dev_query(graph, **?*x*.shortest_path(*x*)**) which transforms an explicit route description like the one above into the following:

```
[leg('5', 'torp', 'centralstationen')]
```

The last action dev_do(map, 'DrawRoute') calls the map agent responsible for displaying the route on the map.

The plan **?*x*.shortest_path(*x*)** described above alone covers all functionality of the GOTTIS application. It should be emphasized that this single plan is sufficient to provide the system with the advanced GoDiS dialogue management described in Section 2.1.1. Examples of this will be shown in the interactions described in Section 2.3.3.

[7] Strictly speaking, the **top** plan is superfluous in this simple prototype application. However, it has been included for compatibility with other GoDiS applications, and to enable easy extension of application capabilities.

Device resources The device resource of GOTGODIS consists of a graph device and a map device. The graph device is used to retrieve the shortest route between two stops. It can execute 2 queries:

- **?*x*.shortest_path_expl(*x*)**
- **?*x*.shortest_path(*x*)**

The first query is performed as an OAA solvable sent to a Graph agent. The query is executed by the perform_query method shown below:

```
perform_query( X^shortest_path_expl(X), shortest_path_expl(Shortest_path) ) :-
dev_get( dept_stop, Dept_stop ),
dev_get( dest_stop, Dest_stop ),
tkit_oaa:solve(shortest_path(Dept_stop, Dest_stop, Shortest_path)).
```

The second query transforms the explicit route description stored in the systems private beliefs to a more usable representation and stores it in its private beliefs. The perform_query method associated with this query is shown below:

```
perform_query( X^shortest_path(X), shortest_path(Path_term) ) :-
dev_get( shortest_path_expl, Shortest_path_expl ),
create_godis_repr(Shortest_path_expl, Path_term).
```

The map device is responsible for drawing the route description on the map. It can execute a single action, DrawRoute, resulting in an OAA solvable being sent to the Map agent.

Dialogue examples

In this section we present sample interactions with the GOTGODIS application which show examples of how the inherited dialogue management of GODIS has made the application both robust and flexible.

Example interaction 1 The purpose of this interaction is to show that interactions possible in GOTTIS are also still possible in GOTGODIS. The user specifies all required information in one utterance.

(13) S: What can I do for you

U: I want to go from Angered to here [clicks on biskopsgarden]

S: One takes tram number 8 from angered to gamlestadstorget, tram number 7 from gamlestadstorget to centralstationen, bus number 60 from centralstationen to brunnsparken and then tram number 2 from brunnsparken to biskopsgarden.

the route is drawn on the map

Example interaction 2 This interaction shows accommodation of an unraised issue (departure stop), negative perception feedback from the user, and belief revision. The user first specifies chalmers as the departure stop but then later changes his mind and says ”from brunnsparken”.

(14) S: What can I do for you?
U: I want to go from here [clicks on chalmers]
S: Okay. Let us see. Which stop would you like to go to?
U: pardon
S: Okay. Let us see. Which stop would you like to go to?
U: from brunnsparken
S: Okay. So, which stop would you like to go to?
U: angered
S: One takes bus number 60 from brunnsparken to centralstationen, tram number 7 from centralstationen to gamlestadstorget and then tram number 8 from gamlestadstorget to angered.
the route is drawn on the map

Example interaction 3 This interaction shows accommodation, grounding, and feedback. The system, given the answer **departure_stop(biskopsgarden)**, accommodates the dependent issue of **?*x*.shortest_route(*x*)**. The number `0.6` behind the utterance ”from biskopsgarden” specifies a low recognition score and as a result of this the system provides interrogative feedback (on the “contextual understanding” level). Also we see how the system rejects the proposition **dest_stop(solrosgatan)** as this is not a valid stop in the domain.

(15) S: What can I do for you?
U: from biskopsgarden [0.6]
S: Alright. You want to know what the shortest route is.
U: yes
S: Let’s see. You want to go from biskopsgarden, is that correct?
U: yes
S: Okay. Which stop do you want to go to?
U: to solrosgatan
S: You want to go to solrosgatan. Sorry, solrosgatan not a valid stop. So, Which stop do you want to go to?
U: [clicks on bergsjon]
S: One takes tram number 2 from biskopsgarden to brunnsparken, bus number 60 from brunnsparken to centralstationen and then tram number 7 from centralstationen to bergsjon.
the route is drawn on the map

Example interaction 4 This fourth example interaction first shows how the user switches from English to Swedish and then how the system handles the ambiguous utterance ”chalmers” by asking the user a clarification question of whether he wants to go from chalmers or to chalmers (”Vill du åka från chalmers eller vill du åka till chalmers?”).

(16) S:What can I do for you?

U:[clicks on a button for Swedish language]

U: chalmers

S: Chalmers. Jag förstår inte riktigt. Visst. Du vill veta vad som är den kortaste rutten.

S: *Eng. Chalmers. Sorry, I don't quite understand. Alright. You want to know what is the shortest route.*

U: ja

U: *Eng. yes*

S: Få se. Vill du åka från chalmers eller vill du åka till chalmers?

S: *Eng. Let's see. Do you want to go from chalmers or do you want to go to chalmers?*

U: från chalmers

U: *Eng. from chalmers*

S: Okej. Vilken hållplats vill du åka till?

S: *Eng. Okay. Which stop do you want to go to?*

U: torp

U: *Eng. torp*

S: Man tar spårvagn nummer 6 från chalmers till vasaplatsen, spårvagn nummer 3 från vasaplatsen till valand, spårvagn nummer 4 från valand till korsvägen och sedan spårvagn nummer 5 från korsvägen till torp.

S: *Eng. You take tram number 6 from chalmers to vasaplatsen, tram number 3 from vasaplatsen to valand, tram number 4 from valand to korsvagen and then tram number 5 from korsvagen to torp. the route is drawn on the map*

2.3.4 AGENDATALK

AGENDATALK is a multimodal and multilingual dialogue application built with the GODIS dialogue manager and the TRINDIKIT toolkit as a spoken interface to a freely available schedule management software called the BORG Calendar. The user can ask AGENDATALK about items noted in the calendar e.g., "What time is the meeting?" as well as instruct the system to take down notes e.g., "Add a meeting the 6th of October at 17". The calendar can also be accessed through the graphical interface like in a standard desktop calendar application in the in-home environment.

Scenario

AGENDATALK works as a voice interface to a graphical calendar in the in-home environment. The user can access her calendar through spoken dialogue and has the possibility to follow the updates made in the calendar on the screen. Imagine the following scenario where we find our user relaxing on the sofa listening to some music:

(17) *The mp3 player DJGoDiS is playing some music. The phone rings*

USR> Pause the music

User takes the call. A friend is calling to see if they can get together at the end of March when she's in town. User tells her friend to wait a minute and turns AGENDATALK *on.*

USR> What do I have the 13th of March?

The calendar interface jumps to March and highlights the bookings that day. User turns back to her friend on the phone and confirms an appointment. Finishes the call and turns back to the agenda.

USR> Add dinner that day at seven pm.

SYS> OK. Dinner the 13th of March at seven pm?

USR> Exactly.

SYS> Scheduled.

The booking turns up highlighted on the screen

USR> What time is the dinner tonight?

SYS> At nine pm.

System goes back to the current month and highlights the time of the dinner event. User takes a glance at her watch, takes a seat in the sofa and addresses DJGoDiS

USR> Turn on the music again

Music goes on again

A talking calendar is not only useful in the in-home environment but could be a preferred choice of interaction with your calendar in the in-car environment or as interface to the calendar on a mobile device. However, the behaviour of the AGENDATALK system would need to be adapted to the different environments dependent on how much possibility the user has to follow the graphical output.

Infrastructure

AGENDATALK has been built with the latest GODIS version and TRINDIKIT4. It uses Nuance for ASR and Vocalizer and Realspeak for TTS. It uses a Nuance wrapper agent and some extra OAA agents for text output and input for the text version.

Figure 2.5 shows the AGENDATALK architecture. The database resource is a MySQL calendar database connecting AGENDATALK with the graphical calendar application BORG by sharing the same calendar information. A wrapper for the BORG Calendar, the BORG Agent, has been built to be able to communicate directly with the graphical interface to enable multimodal output. The AGENDATALK system also uses some additional GODIS modules to handle multimodal fission. These are described in detail in [9] and are therefore not included in the architecture here, and are only briefly mentioned below.

Research issues addressed

AGENDATALK makes use of the latest enhancements of the GODIS system (e.g., dialogue move confidence) and the new asynchronous TRINDIKIT4 which enables barge-in both via speech and via graphical input. It uses SLMs generated from GF grammars according to the methodology in Task 1.3 and makes use of the GODIS option of switching language on the fly. WP3 research on multimodal fission and content reduction has been integrated into AGENDATALK to make it possible to distribute the output over modalities in a more sophisticated way. A simple implementation of reference resolution has been added

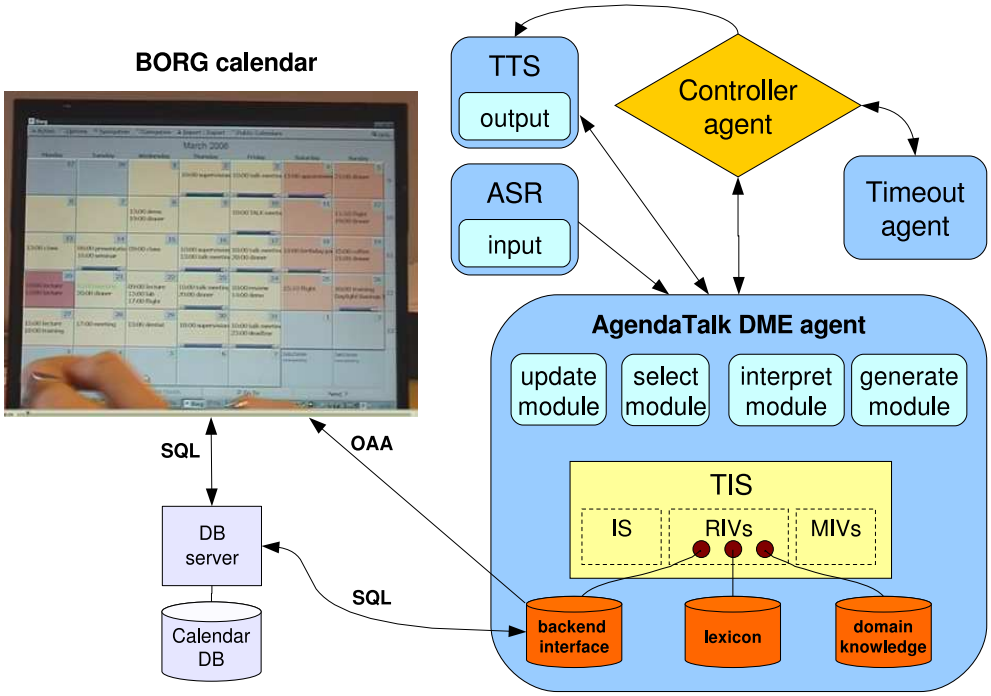

Figure 2.5: The AGENDATALK Architecture

to make it possible to refer to previous mentioned bookings. AGENDATALK makes use of a TRINDIKIT-specific logging format to log dialogue context to collect data for future dialogue system research. All this is enabled by an extended information state that holds information used for multimodal fission and reference resolution and that keeps track of the dialogue history for more complete logs.

Functionality

A graphical interface for a calendar may seem superior as an interface mode compared with a speech interface in the in-home environment. In the special situation of the in-car environment or on a mobile device the advantages are clearer. However, we would like to point out that there are scenarios where the spoken dialogue can be superior to the graphical one and very practical e.g., when searching for bookings not shown on the screen (i.e., the current month/week) where the user would have to go through every month while AGENDATALK could give you the answer directly. In the same way AGENDATALK could give you the information if you are booked or not a certain date without having to step through the calendar until that date. Apart from this, there are many situations in the in-home environment where speech is a better interaction choice than keyboard input and mouse clicking. Importantly, the availability of both a spoken and a graphical interface gives the user the advantages of both modalities.

The AGENDATALK application supports the following functionality which makes it possible for the user to both change her schedule, check her bookings or navigate the graphical calendar:

- Add a booking
- Reschedule a booking (date and/or time)
- Delete a booking
- Delete a whole day's bookings
- Ask for the time of a booking
- Ask if booked a certain time or date
- Ask about bookings a certain date
- Ask for today's date
- Ask for the date of a booking
- Ask for bookings on a certain part of the day
- Go to a specific date in the calendar
- Go to a specific month in the calendar
- Change language of the dialogue and the Calendar

Multilinguality

AGENDATALK works in Swedish and English and the user can switch between these on the fly whenever she wants, by giving a switch language command. The system will then switch grammars, language models, ASR, TTS and change the language of the BORG Calendar. However, the calendar information will be kept as it is. After a language switch the system will continue in the same state of the dialogue. This is possible due to the modularity of GODIS which keeps all dialogue management parts language independent. The only language dependent parts are the actual grammars and language models for ASR.

For Swedish, AGENDATALK uses an SLM generated from the AGENDATALK GF grammar using the methodology in Task 1.3 and for interpretation we are using a Prolog Lexicon. Nuance is used for speech recognition and Realspeak is used for Swedish TTS. For English, AGENDATALK uses an SLM generated from the English version of the AGENDATALK grammar and for interpretation a Prolog Lexicon. Nuance and Vocalizer are used for ASR and TTS. The language switch of the calendar is done by sending an OAA solvable to the BORG Agent.

AGENDATALK would probably normally be used by the same user in one single language. The switching has mainly been implemented to show the modularity of the GODIS system. However, for bilingual speakers it is very easy to imagine bilingual scenarios such as the following one where a bilingual AGENDATALK would be useful:

(18) *User, native English speaker, is talking with a Swedish colleague*

USR> Ska vi träffas nästa gång kl 15 på mån

Eng. Can we meet next time on Monday at 15?

Colleague>Okej

Eng. OK

USR> (to the system) Lägg till ett möte kl 15 på måndag 13 november

Eng. Add a meeting at 15 on Monday 13th of november

SYS> bokat

Eng. Scheduled

Colleague gets up to go

USR> Hejdå

Eng. Bye

Colleague> Hejdå

Eng. Bye

USR> (to the system) Byt till Engelska

Eng. Switch to English

Calendar screen changes to English

SYS> What do you want to do?

USR> Change the yoga session on Monday to Tuesday

SYS> Tuesday at 15?

USR> Correct

SYS> Rescheduled

Multimodality

AGENDATALK can be run in three different modes: graphics-only (i.e., using the standalone BORG Calendar), multimodal mode, and speech-only mode. The multimodal mode consists in the user choosing to give input via voice and being able to see the changes in real-time made in the calendar on a screen. The system will keep the user informed of its actions both with speech and through the graphical interface. The user also has the possibility of giving graphical input in the form of clicking on days in the calendar to inform the system of which date is considered. This can be done in a separate turn or combined with speech in an integrated multimodal turn. The third mode is the case when no screen is available or in the case the user has less possibility to follow updates on the screen e.g., while cooking (or in the in-car environment). In this case the system would give all necessary information via speech.

Research on content reduction and multimodal fission [9] has been integrated into AGENDATALK which makes it possible to distribute different parts of a system message over different modalities, in this case the graphical calendar and the spoken output. This means that AGENDATALK can decide how and where to present the informative part of a message, i.e., the focus, and the backgrounded part, the ground. Each modality can either realise both the focus and the ground, just the ground, just the focus or nothing at all. In this way, AGENDATALK generates multimodal contributions that help the user to know what to focus on in her schedule. The graphical output is made by highlighting or flashing calendar information in different colours to produce focus and ground. The speech output realises focus by using SSML emphasis tags for the TTS.

The graphical calendar system chosen to be used with AGENDATALK is the calendar system BORG (http://borg-calendar.sourceforge.net/) which is an open-source calendar application written in Java. To enable multimodal fission we have developed an OAA wrapper for the BORG system to be able to control the graphical interface in a better way (see D3.3) e.g., to highlight some information in the calendar. AGENDATALK is also connected with the BORG system by altering the same MySQL database and in that way share the same information. The AGENDATALK application interacts with the database via the SQL Prolog API agent described in D5.1.2 [3]. Whenever anything is changed in the database, AGENDATALK will tell the BORG Agent via OAA to update the graphical interface.

In addition, an OAA wrapper processing iCal files, the iCal Agent, has been developed not to close doors for other calendar applications to be connected with AGENDATALK (see D5.1.2 [3]).

Implementation of application specific resources and modules

AGENDATALK has four resources apart from the lexicon/grammar resources. AGENDATALK's domain resource holds the domain knowledge with the domain plans that directs the dialogue. Apart from this, AGENDATALK holds calendar knowledge gathered in a separate resource. The device resource manages the communication with the database resource and the BORG Agent. AGENDATALK also has some additional GODIS modules to be able to handle multimodal fission. This section will describe all these application specific resources and modules.

Calendar Resource The calendar knowledge holds all knowledge about dates and includes deictic information and information about inconsistent dates. It gives the number of a month or day e.g., that March is month number three of the year. The calendar resource also helps AGENDATALK to know which dates are valid and which are not: e.g., that February has 29 days some years, that April just has 30 days and thereby that the 31st of April is not a valid date. If the user gives a date that is not valid, the

system will point this out and ask for the date once again. Apart from this, the calendar resource supplies AGENDATALK with information about what date an expression such as "next Friday" refers to. It also holds information about times and gives back disambiguated times e.g., that 4.00 pm means 16.00 in a 24-hour notation.

Domain Resource In AGENDATALK we have the following 15 domain plans to direct the dialogue:

- **top**
- **alter_calendar**
- **get_info**
- **add_event**
- **delete_event**
- **delete_current_event**
- **delete_all_events**
- **change_event**
- **change_language**
- **goto**
- **usage**
- **?***x*.**time(***x***)**
- **?***x*.**date(***x***)**
- **?***x*.**bookings(***x***)**
- **?***x*.**todaysdate(***x***)**

As can be seen, we have two different types of plans: the action-oriented ones (where the user requests an action to be performed), and the issue-oriented ones (where the user asks a question) which are the last four ones.

We will start by specifying the plans dealing with actions. The **top** plan is loaded at start-up and looks as follows:

ISSUE : top

$$\text{PLAN:} \left[\begin{array}{l} \text{forget\_all} \\ \text{raise(}\textbf{?}x.\textbf{action(}x\textbf{)}\text{)} \\ \text{findout(} \left\{ \begin{array}{l} \textbf{?action(alter\_calendar),} \\ \textbf{?action(get\_info)} \end{array} \right\} \text{)} \end{array} \right]$$

This plan has as its only action to find out what action the user wants to perform or what issue she wants to raise. This action is realised as the question "What do you want to do?". In AGENDATALK, the choices would be either to alter the calendar or to get information of a specific event

noted in the calendar. This would correspond to the more specific phrase “Do you want to change something in the calendar or check your bookings?” which will be generated if the user does not answer the more generic question above. The dialogue will always return to this top plan when other actions/plans have been finished and the system will say something like “Returning to top”. The forget_all action is used to call an update rule that clears the information state which includes deleting e.g., all propositions in private beliefs and all mutually agreed propositions in shared commitments to prepare itself for new actions or issues.

The following two plans in the list deal with the two different high-level options i.e., alter the calendar or get some information in a more specific way. If the user requests to alter her calendar, the system will ask what kind of action she wants to perform of the following: add a booking, delete a booking, reschedule a booking or navigate the calendar interface by jumping to a month or date. However, if the user is not a novice she would probably go directly from the topmost plan to the specific plan she wants by raising the correct action e.g., adding a booking. These plans exist to guide novice users stepwise.

A more advanced plan in AGENDATALK is the plan corresponding to the **add_event** action.

```
ACTION : add_event
        [ findout(?x.event_to_store(x))
        | findout(?x.date(x))
        | findout(?x.start_time_to_store(x))
        | findout(?x.am_or_pm(x))
        | dev_query(agenda,?booked)
        | if booked
        | then [ findout(?doublebook)
        | else [ findout(?take_down_event)
        |
PLAN:   | if take_down_event
        |       [ dev_do(agenda,AddEvent)
        | then | report(AddEvent,done)
        |       [ forget_all
        |       [ if doublebook
        |       |       [ dev_do(agenda,AddEvent)
        | else  | then  | report(AddEvent,done)
        |       |       [ forget_all
        |       [ else  [ report(AddEvent,failed(doublebook))
        [                 [ forget_all
POSTCOND : done(AddEvent)) || status(AddEvent,failed(_))
```

This plan consists of findout plan constructs to get the required information from the user, to be able to add a booking in the calendar database. The system needs to find out the event, the date, the time and to disambiguate what day-half the time corresponds to (i.e. AM or PM) to be able to note down an event in the calendar. The findout plan constructs correspond to ask moves in the lexicon which are then translated to natural language like in the following example:

```
findout(X^event_to_store(X))
```

```
output_form(ask(X^event_to_store(X)),['What kind of event are we talking about?']).
```

When all the information has been gathered (either by the user providing the information stepwise or in one single turn) the next action is to check with the calendar whether the user is booked or not at the agreed time slot before adding it to the calendar. This is done by querying the database with the dev_query(agenda,booked) action. If the user already has a booking at the agreed time slot (i.e. the condition booked is fulfilled) the system should make the user aware of the possible double booking and verify if the user really wants to doublebook. Otherwise, if the time slot is available, the system needs to find out if the user wants to take down the information understood (i.e., the event, the date, the time) using the simple findout construct **findout(take_down_event)** which corresponds to a yes/no question. If that is the case, the dev_do action communicates to the agenda device resource to perform an AddEvent action i.e., add the booking to the calendar database. If it succeeds, the user will be informed that the information has been taken down. In the case the user is informed of a possible double booking the user has the option of adding the booking anyway. Otherwise, the system will not alter the calendar. Finally, a postcondition is added to the plan. The postcondition checks when the action (i.e., the plan) AddEvent has been completed successfully (i.e. introduced in the calendar database). The GODIS update rule **exec_dev_do** adds to the private belief in the information state that an action has been done after it has been performed. This is what is checked with the postcondition to assure that the action has actually been performed. For the following plans we will exclude the postcondition descriptions as they work in a very similar way and is not of particular interest.

The **delete_event** plan is the plan for deleting a booking from the calendar. To be able to do this we minimally need to find out the booking and the date of the booking. If the user specifies a time it should only look for bookings at that time (in the case there exist various similar bookings the same date). The **bind** construct in the plan binds information that the user may supply but are not required to and will therefore not trigger any findout actions. This is the case of time specification. This implies that the system will never ask for the time of a booking to be able to perform a delete action but in the case the user supplies this information it will be used in the search of the event to delete. The last part of the plan holds the actual delete action dev_do(agenda,DeleteEvent) which calls the device resource to perform a deletion of an event.

ACTION : delete_event

PLAN:
- findout(**?**x.**event_to_store(**x**)**)
- findout(**?**x.**date(**x**)**)
- bind(**?**x.**start_time_to_store(**x**)**)
- bind(**?**x.**am_or_pm(**x**)**)
- dev_do(agenda,DeleteEvent)

POSTCOND : **done(DeleteEvent)** || **done(DeleteEvent,failed(_))**

The **delete_current_event** plan makes it possible to delete all the information the system has gathered about an event during a dialogue in case it does not coincide at all with what the user wanted to take down. The **delete_all_events** plan enables the functionality to delete all bookings a specific date. This means that the user can perform actions such as “delete all bookings on Monday”. The only required information to be able to do this is a date.

The **change_event** plan handles rescheduling of bookings either by changing the time of an event or moving a booking from one day to another. This is a very nested plan due to all the possibilities in rescheduling. To be able to perform a change of a booking the system minimally needs to know what booking should be rescheduled and what date it is booked at the moment. Apart from this, the system needs to find out what kind of change the user wants to perform i.e., rescheduling the time, the date or both. If the user gives any clues in her request of what type of change it is the system should use these clues to direct the dialogue correctly. For example, if the user says "change the meeting on Friday to Monday" the system should draw the conclusion that this is a change of date as the user is giving a new date and change the date of the booking. However if the user says "reschedule the meeting on Friday" the system needs to find out what changes the user wants to perform and find out the necessary information e.g., a new date or a new time. The user also has the possibility of changing both the date and the time of a booking which will result in two different device actions ChangeTime and ChangeDate.

The **goto** plan makes it possible to navigate the graphical calendar by jumping forward and backward between months or go to a specific date or month in the calendar. If the user requests the system to go to a specific date the device action GoToDate will be activated. In the case the user wants to go to another month the GoToMonth action will apply. Otherwise, if the user does not specify date nor month (e.g., by saying just "jump in the calendar") the system will ask for a date and apply the GoToDate action when the date has been specified by the user.

The **change_language** plan enables the language switch explained in the section on multilinguality. This plan will get activated on user initiative with requests such as "Change the language". To be able to perform a language switch the system minimally needs to find out what language to switch to. This corresponds to the construct findout(**?***x*.**language(***x***)**). The **change_language** construct on the plan will trigger an update rule which changes the LANGUAGE variable in the information state. This will then trigger a change of language resources such as grammars, ASR, and TTS. The dev_do(agenda,ChangeLanguage) construct forces the device resource to execute the **change_language** command which sends an OAA solvable to the BorgAgent requesting a change of language in the graphical interface.

ACTION : change_language
PLAN: [findout(**?***x*.**language(***x***)**)
change_language
dev_do(agenda,ChangeLanguage)]
POSTCOND : **done(change_language)** || **done(ChangeLanguage,failed(_))**

The **usage** plan comes into action if the user asks for help explaining the usage of the AGENDATALK application.

The last four plans on the list above correspond to user queries about scheduled events. These are the issue-oriented plans. The first one simply finds out today's date by querying the device resource. The next plan treats queries about the time of a booking.

ISSUE : **?**x.**time(**x**)**

PLAN:
- findout(**?**x.**event_to_store(**x**)**)
- bind(**?**x.**date(**x**)**)
- if date(_)
- then
  - dev_query(agenda,**?time(_)**)
- else
  - dev_query(agenda,**?date(_)**)
  - dev_query(agenda,**?time(_)**)

It minimally needs the type of booking and the date of the booking to query the device resource for the time set for the booking. The date information is optional by using a bind construct. This means that if the user provides this information it will be used to look for the time of an event the specified day. In the other case, the system will first query the device (i.e. the calendar) about the date of the user-specified event and then search for the time. The time search is done by querying the device with the **dev_query(agenda,?time(_))** command. How a device query works is described in the device resource section.

In the case the user wants to find out the date of a booking the system minimally needs to know what type of event it is. In the case the user supplies additional information such as the time of the event the device query will search with these conditions. This behaviour is captured in **?**x.**date(**x**)** plan. The last plan in the list **?**x.**bookings(**x**)** takes care of user queries of what is scheduled a certain date or time. It minimally needs the date but if the user supplies additional information such as a specific time or a part of the day (e.g., in the afternoon) the system will restrict its search with these parameters. Again a query is sent to the device resource which makes use of the database resource to convert the query into SQL and search the calendar database shared with the BORG Application.

Apart from these plans we have added commands to the device resource that performs the device actions and queries appearing in the plans by interacting with the calendar database and the Borg Agent as described in the following section.

Device Resource The device resource is the interface to the database resource interacting with the SQL Prolog API Agent (see D5.1.2 [3]) and also the interface to the BORG Agent. In this resource, the (non-communicative) actions and queries that the device will perform have been defined, with associated arguments. In addition, a routine for executing the device actions has been added. The device resource can execute 16 different actions performed as database commands or OAA solvables sent to the BorgAgent.

- action AddEvent
- action DeleteEvent
- action DeleteAllEvents
- action ChangeTime
- action ChangeDate

- action GoToDate
- action GoToMonth
- action ChangeLanguage
- action StartCalendar
- action ColorEvent
- action ColorDay
- action ColorTime
- action FlashEvent
- action FlashDay
- action FlashTime
- action SetToDefault

In addition, it can execute 7 queries to the database or the calendar resource.

- query **?.*x*time(*x*)**
- query **?.*x*bookings(*x*)**
- query **?.*x*todaysdate(*x*)**
- query **?.*x*date(*x*)**
- query **?.*x*datetime(*x*)**
- query **consistent_date**
- query **booked**

The device commands and queries are controlled partly from the plan but also from the generation module to control the graphical interface for multimodal fission. The plans in the domain resource correspond here, in the device resource, to goal actions. For example, the **DeleteEvent** plan in AGENDATALK's domain knowledge corresponds to the goal action DeleteEvent defined with its corresponding parameters as follows:

```
action( 'DeleteEvent',[event_to_store, date]).
```

In the device resource, the values of the device parameters will be set by taking the values of the shared commitment propositions in the information state /SHARED/COMMITMENT, and interpret them pragmatically. This is in order to handle dates in a more specific way such as converting the piece of information **date** with the value 'today' into today's date (e.g., `date(2006,12,25)`). This is done using the calendar resource mentioned earlier.

The next step is to get the device to perform the current action e.g., to delete an event, by first getting all the values of the parameters needed to delete a booking in the calendar database. The information is then sent to the database agent. The **perform_command** for the DeleteEvent action realises the command and calls the database agent **database_agenda** to alter the calendar database and sends OAA solvables to the BORG calendar to update the graphical interface.

In a similar way the device resource performs the actions: AddEvent, DeleteAllEvents, ChangeTime and ChangeDate by interacting with the database resource. The actions: GoToDate, GoToMonth, ChangeLanguage and StartCalendar do not alter the database but are performed by sending OAA solvables to the BORG Agent which will then realise the corresponding action. The **perform_command** for the ChangeLanguage action exemplifies this:

```
perform_command( 'ChangeLanguage'):-
        !,
        dev_get(language,Lang),
        lang2borglang(Lang,BorgLang),
        tkit_oaa:solve(change_language(BorgLang)).
```

We can see that the LANGUAGE variable needed will be picked up from the information state with the **dev_get** method and converted into the format BORG has for languages (using the **lang2borglang** predicate) and thereafter the OAA solvable change_language will be sent using TRINDIKIT OAA facilities. The Borg Agent will then force the BORG Calendar to switch GUI language.

The other remaining actions on the list (e.g., ColorTime) are the ones controlling the colouring of the graphical calendar interface used for multimodal fission. These also interact with the BORG Calendar via OAA as described in [8].

In the case of querying the schedule in the calendar database, the **device resource** will take a query and construct a command with the type of search and the corresponding parameters needed to call the **database** agent. However, for some general queries we will call the **calendar knowledge** resource. The seven queries enumerated above correspond to: verifying if a date is consistent using the calendar knowledge resource, searching the calendar database for what bookings are scheduled a certain date or time, searching the database for the time scheduled for an event, searching the database for the date of a booking, finding out if the user is booked to be able to inform of scheduling conflicts and finally querying the calendar knowledge resource of today's date. An example of how such a query is performed (in this case querying the database for the time of a booking) can be found below:

```
perform_query(time(Time),time(Time)):-
        !,
```

```
dev_get(event_to_store, E),
dev_get(date, D),
date2dbdate(D,DBDate),
database_agenda:selectDB(_Table,time,set([text=E,appt_date>>DBDate]),
                         appt_date=[DateTime|Rest]),
(DateTime == empty, Time = empty
;
dbdate2datetime(DateTime,Date,Time)).
```

The query is performed by getting the type of booking and the date (and converting the date to the specific calendar database format using the **date2dbdate** predicate) and thereafter call the **database** resource (i.e., the SQL Prolog API) with the **selectDB** command to convert the query into a SQL SELECT command. Finally, the **database** resource will call the MySQL database to get the time for the booking at the given date. Again, a conversion from the obtained database date and time format is carried out to get a GODIS time proposition (using the **dbdate2datetime** predicate). In the case the system does not find any event booked the given date, the system will inform of the absence of this data.

Database Resource The database used is a MySQL database holding the calendar information that appears in the Graphical Interface. The database can be altered either directly by using the graphical interface BORG (e.g., delete an event by selecting it and deleting it) or by using the dialogue system and managing the calendar via voice. Changes coming from either source will be reflected in the other mode.

We have developed a simple SQL interface written in Prolog to connect AGENDATALK with the MySQL calendar database. This interface is used to be able to alter or query the graphical calendar application's database table. The SQL interface takes Prolog lists and constructs SQL commands from these, calls the MySQL database and finally constructs an appropriate answer in return.

The interface offers five basic functionalities:

- search for answers to queries
- add items to the database
- delete items from the database
- count number of items having certain values
- update values of items in the database

The database agent is described in more detail in D5.1.2 (see SQL Prolog API) [3].

Information structure Module and Multimodal Fission Module In order to enable a flexible system behaviour with regard to content reduction and media allocation, two new modules have been added to AGENDATALK: an **Information Structure** module and a **Multimodal Fission**

module. The **Information Structure** module operates on the output given by the **Select** module, that is, on the next moves, and their contents, to be generated by the system. The **Information Structure** module determines the information structure of these move contents, dividing the pieces of content into focus (the "new", informative part, as explained above) and ground (the backgrounded, already known part, as also explained above).

The **Multimodal Fission** module operates on the information structure-determined contents given by the Information Structure module. Based on a number of contextual factors, this Fission module determines whether a content reduced contribution is to be realised, that is, whether just the focus or both focus and ground are to be realised. It also determines media allocation, that is, which parts of the contribution are to be realised in this modality.

The **Generation** module has also been modified to handle information structure. Pieces of content marked as focus and to be realised using speech, are here mapped to an SSML tag giving emphasis (see example (19) below).

These three modules are described in detail in [9].

Dialogue examples

The following illustrative example dialogues from interactions with AGENDATALK shows the multimodal and multilingual behaviour and some functionality of the application.

Example interaction 1 The following dialogue shows the multimodal fission behaviour of AGENDATALK and some simple reference resolution.

(19)
USR> i want to add a dinner next Friday at seven pm
SYS> Do you want to take down: dinner Friday the 1st of December at 19?
USR> Correct
SYS> Scheduled
Booking appears on screen highlighted
SYS> What do you want to do?
USR> What time is the dinner next Friday?
SYS> The dinner is at <emphasis> 19 </emphasis>
The content word with the time of the dinner is blinking on the screen, the dinner event is highlighted and the TTS puts emphasis on the time using SSML
USR> Am i booked at seven pm next Friday?
SYS> You have a <emphasis> dinner </emphasis> that time.
The content word with the event is blinking on the screen, the time is highlighted and the TTS puts emphasis on the event using SSML
SYS> What do you want to do?
USR> Change the time of it to eight pm

The system changes the time of the last considered booking i.e., the dinner the 1st of December

SYS> Rescheduled

Example interaction 2 This example dialogue shows both topic and language switch in AGENDATALK as well as how multimodal input can be intertwined in the dialogue.

(20)
SYS> What do you want to do?
USR> i want to add a booking
SYS> What kind of booking?
USR> a meeting
SYS> OK. What date?
USR> [clicks on a date (Dec 10)]
SYS> What time is the meeting?
USR> am i booked at ten am?
SYS> you have no bookings at that time
SYS> So. What time is the meeting?
USR> Switch to Swedish
System switches language and switches the language of the calendar interface
SYS> OK. Vilken tid är mötet?
Eng. OK. What time is the meeting?
USR> Klockan 9 på morgonen
Eng. At 9 am
SYS> Vill du lägga till: möte den tionde december klockan 9?
Eng. Do you want to add: meeting the tenth of december at 9 am?
USR> jajamen
Eng. yes
SYS> Scheduled
USR> Byt språk till engelska
Eng. Change language to English
System switches language and switches the language of the calendar interface
SYS> OK. What do you want to do?
USR> Delete the meeting [clicks on the 10th of December]
The system deletes the meeting the specified day by the click and it disappears from the calendar screen
SYS> Deletion done

2.3.5 DJ-GODIS

The DJ-GODIS application is a multilingual multimodal interface to an MP3 player, where the user can use a combination of spoken natural language and pointing gestures to create and manipulate a playlist, play songs from the playlist, control the volume and query the music database in different ways.

Two MP3 player systems have been developed in TALK, DJ-GODIS being one and SAMMIE [1] the other. The motivation for having two systems is that they explore different research issues, SAMMIE the in-car domain and multimodal output strategies, among other issues, and DJ-GODIS notably the in-home domain, the integration of multimodality and multilinguality, and multimodal menu-based dialogue.

Scenario

In the following scenario, the user wants to listen to some music.

(21)
USR> Play Massive Attack
SYS> OK. What song do you mean?
USR> What songs do I have?
SYS> Angel, Sly and Teardrop.
SYS> Returning to playing a song, which song do you want to play?
USR> Teardrop
Music starts playing
USR> Add Sly to the playlist too
SYS> Okay. I have added a song to the playlist.

Infrastructure

The application uses the GODIS dialogue manager and the TRINDIKIT4 dialogue system toolkit and consists of a collective of OAA agents organized as in figure 2.6. The Controller agent, DME agent and the MMD, ASR and TTS agents are Trindikit agents, which communicate using the TRINDIKIT4 OAA API described in TALK deliverable D5.1.2 [3].

- The controller agent coordinates the different modules and agents by executing a set of serial control algorithms in parallel.
- The timeout agent is used by the controller to determine when the user's turn is over.
- The DME agent holds the total information state (TIS) and the core dialogue management modules, update and select, as well as interpretation and generation modules.
- The actual interpretation and generation is done by the GF agent, which is called over OAA by the interpretation and generation modules.

- The DynGUI input/output module agent is used to dynamically render graphical menus which can be used for graphical input.
- The ASR module agent continuously listens for input and writes the recognized result to TIS.
- The TTS module agent reads output from TIS and synthesizes it as speech, when called from the controller.
- The MP3 GUI agent displays a graphical representation of available songs and the current playlist as two lists, and offers the possibility for the user to make graphical input by clicking on the list items. It also eavesdrops on the commands sent to the MP3 player agent and updates its playlist representation whenever a relevant MP3 player solvable is called.
- The MP3 player agent plays music files.

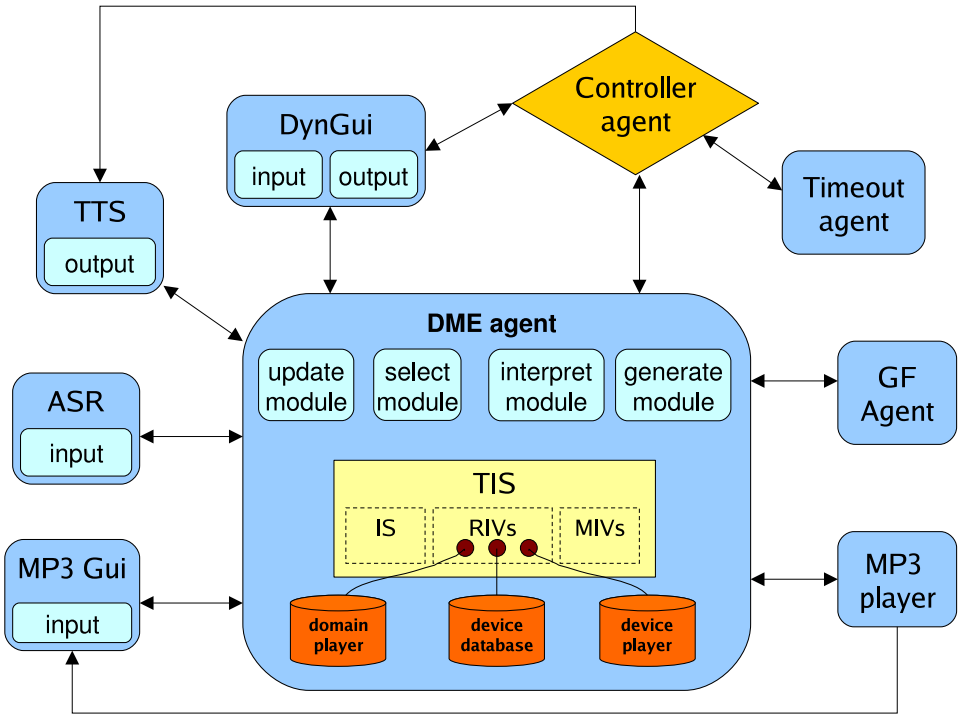

Figure 2.6: DJ-GODIS system as a collective of agents

Research issues addressed

DJ-GODIS addresses the following TALK research issues:

Integrated approach to multilinguality and multimodality By deploying multimodal (and multilingual) GF grammars, DJ-GODIS demonstrates the integrated approach to multimodality and multilinguality described in Section 2.2 as well as in D1.2b [15] and D1.5 [14].

Multimodal menu-based dialogue DJ-GODIS demonstrates the MMD approach to multimodal dialogue management, as described in Section 2.1.2 and D2.1 [18].

Dynamic reconfiguration DJ-GODIS demonstrates implicit application switching and offline plug-and-play, as described in D2.2 [19]. The DJ-GODIS and GODIS-DELUX applications have been implemented independently and no knowledge of the other application is hard-coded into grammars or resources. Nevertheless, they can be run simultaneously with a seamless interface presented to the user.

Asynchronous multimodal dialogue management DJ-GODIS makes use of the asynchronous capabilities of TRINDIKIT4 as described in D5.1.2 [3] to enable multimodal barge-in and incremental interpretation.

Functionality

MP3 players are generally controlled via some sort of menu-based graphical interface. The DJ-GODIS system can be seen as a testbed for the concept of Multimodal Menu-based Dialogue (MMD), where graphical as well as spoken output is generated in parallel from the same abstract representation, and spoken and graphical input can be combined in, and parsed as, one utterance. One of the ideas behind MMD is that the user can choose at any time during the dialogue what modality/ies to use. The generic DynGUI displays dynamically rendered GUI components which are used to perform graphical input. In DJ-GODIS there is also a more traditional application-specific graphical interface showing the songs in the media library and songs in the playlist.
The following functionality is supported by DJ-GODIS, and can be accessed using spoken input, graphical input or a combination of the two:

- add a song to the playlist
- delete a song from the playlist
- clear the playlist
- shuffle the playlist
- play the current song
- stop playing
- play a specific song

- control volume
- ask about available songs and artists

Multilinguality

DJ-GODIS can be used in English and Swedish. The GF grammars used for interpretation and generation are described in Section 2.2.5. For ASR the system uses English and Swedish recognition grammars generated from the corresponding application GF user grammars. At any time during the course of the dialogue the user can change language.

Multimodality

In DJ-GODIS graphical input can come from two sources. The user can either click on dynamically generated buttons in the DynGUI or click on songs in the DJ-GODIS GUI (see figure 2.7 for the MP3 GUI). In both cases the input method is the same, the agent in question appends the string representation of he input to the INPUT_BUFFER TIS variable. Speech and GUI interaction may also be combined in a single utterance.

The system output modalities are speech and graphical output in DynGUI and in the DJ-GODIS GUI.

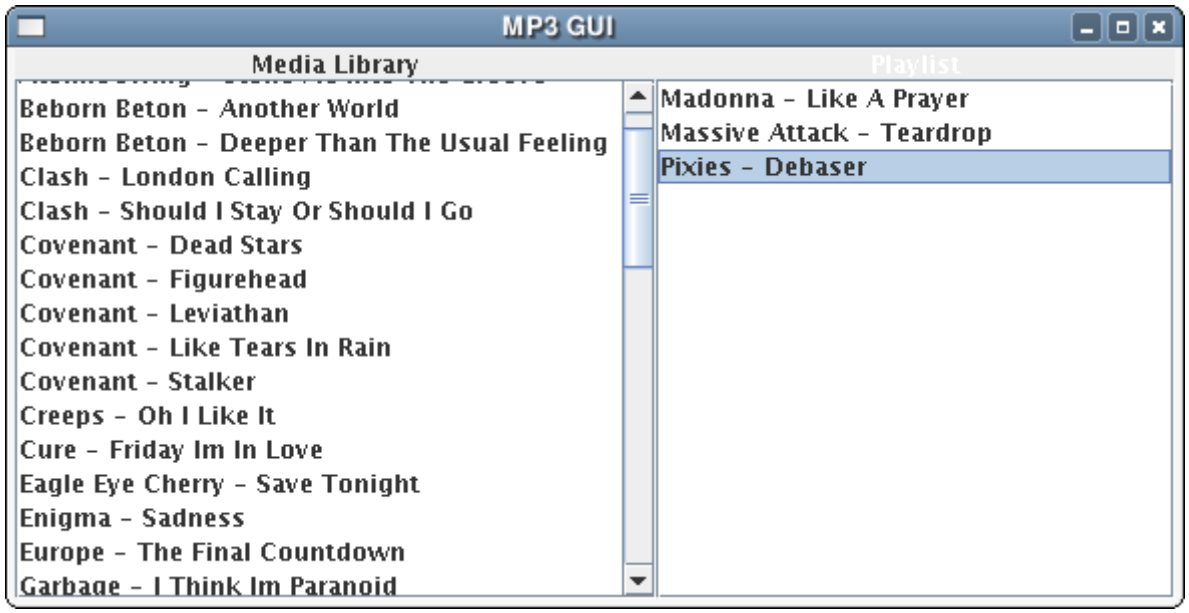

Figure 2.7: The DJ-GODIS MP3 GUI

Implementation of application specific resources

The application specific part of DJ-GODIS consists of three TRINDIKIT resources and the application grammars described in section 2.2.5. The domain resource contains the DJ-GODIS plan library and the domain ontology. The database device resource contains information about the currently available songs. The player device resource is used to communicate with the actual mp3 player, which is a separate OAA agent.

Domain Resource The DJ-GODIS plan library consists of dialogue plans corresponding to the tasks listed in figure 2.8, which also describes the hierarchical menu structure of the plan library.

Hierarchically ordered tasks:

- Top level plan
  - Playback control
    - Play current song in playlist
    - Play a specific song in playlist
    - Stop
    - Fast forward
    - Rewind
    - Control volume
      - Turn volume up
      - Turn volume down
  - Playlist manipulation
    - Add song to playlist
    - Clear playlist
    - Delete song from playlist
    - Shuffle playlist

Tasks outside menu structure:

- Next song
- Previous song
- Query what songs are available by a specific artist
- Query what artist made a specific song
- Basic help

Figure 2.8: DJ-GODIS dialogue plans

An advantage of the MMD approach as that the hierarchical ordering of domain functionality enables naive users to get an overview of the system capabilities, while GODIS accommodation enables expert users to address subtasks of the menu structure directly. We show five of the plans below, each one showing different aspects of GODIS functionality.

The top level plan, which is executed at startup, looks as follows:

ACTION : top
PLAN: [forget_all
raise(**?***x*.**action(***x***)**)
findout({ **?action(control_playback),** **?action(manage_playlist)** })]
POSTCOND : **false**

First the system raises the question "What can I do for you?". The idea of starting with an open question is to not force the user to navigate the menu structure if she does not want to. However, if the initial question is left unanswered by the user, the system will ask the alternative question "Do you want to control playback or manage the playlist". This question is what makes the top level plan act as the topmost node of the task hierarchy, and since its postcondition **false** will never be true, the top level plan will be reraised whenever the /PRIVATE/PLAN IS field becomes empty.

The **manage_playlist** plan is a typical example of a menu node plan. Its only purpose is to guide the user through the menu hierarchy, listing the possible playlist manipulation actions in the form of the alternative question: "Do you want to add a song to the playlist, delete a song from the playlist, clear the playlist or shuffle the playlist?"

ACTION : manage_playlist
PLAN: [findout({ **?action(playlist_add),** **?action(playlist_delete),** **?action(playlist_clear),** **?action(playlist_shuffle)** })]
POSTCOND : **done(playlist_add)** || **done(playlist_delete)** || **done(playlist_clear)** || **done(playlist_shuffle)**

The four different alternative postconditions of **manage_playlist** are those associated with the actions listed as alternatives in the alternative question. The consequence is that when the system has executed one of the subtasks, it has also executed the **manage_playlist** task.

The **play** plan is an example of a simple action plan which communicates with a device. When executing the plan, a **'Play'** command is sent to the player device.

ACTION : play
PLAN: [dev_do(player,Play)
POSTCOND : **done('Play')**

The **playlist_add** plan is a more complex plan which involves communication with both devices. First, the user is asked what song title and artist she wants to add to the playlist. Domain specific inference rules will ensure that if the song title is known, and there is only one matching artist, the system will not raise the artist issue. This also works in a corresponding way if the artist is known.

After figuring out song and artist, the system queries the database device for the path to the music file in question. If the path is known, it sends the **'PlaylistAdd'** command to the player device,

which then puts the file last in the current playlist. If the path is not known, i.e., the file does not exist, the system reports failure and clears information about song title and artist.

ACTION : playlist_add
PLAN:
- findout(**?***x*.**song_to_add(***x***)**)
- findout(**?***x*.**artist_to_add(***x***)**)
- if artist_to_add(*a*)
  - then [dev_set(dbase,artist,*a*)
- if song_to_add(*s*)
  - then [dev_set(dbase,song,*s*)
- dev_query(dbase,**?***x*.**path(***x***)**)
- if path(_)
  - then [dev_do(player,PlaylistAdd)
    - if artist_to_add(*a*)
      - then
        - if song_to_add(*s*)
          - then
            - report(playlist_add,failed(notexist(*a*, *s*)))
            - forget(artist_to_add(_))
            - forget(song_to_add(_))
      - else

POSTCOND : **done('PlaylistAdd')** || **status(playlist_add,failed(_))**

The **?***x***.available_song(***x***)** plan is used for querying the application about which songs are available by a certain artist. If the artist is not known from the context, the system will ask the user what artist she wants to search for. When the answer to the artist issue is known, the system will query the music database device for all available songs by the artist. The device will return a resolving answer and the system will provide the answer to the user.

ISSUE : **?***x*.**available_song(***x***)**
PLAN:
- findout(**?***y*.**artist_available_song(***y***)**)
- dev_queryAll(dbase,**?***x*.**available_song(***x***)**)

Player Device Resource The player device serves as the connection between the domain plans and the actual MP3 player JLGuiAgent, described in deliverable D5.1.2 [3]. The device functionality can be divided into a set of device queries, used for getting information from the device, and a set of device actions, used for changing the state of the device. The execution of device queries and actions can depend on the content of the /SHARED/COM and /PRIVATE/BEL fields, which store the shared commitments and (system's) private beliefs, respectively.

The player device queries are:

- **?***x***.usage(***x***)** – get general help
- **item_to_play_is_in_playlist** – find out whether the song to play is already in the playlist
- **item_to_play_is_in_playlist** – find out whether the song to delete is in the playlist
- **?***x***.index_to_play(***x***)** – find out where in the playlist the song to play is
- **?***x***.index_to_delete(***x***)** – find out where in the playlist the song to delete is

- **?.*x*playlist_item(*x*)** – given a path, find out song title and artist

The player device actions are:

- Play – starts playing the current playlist from its current position
- Pause – pauses playback
- PlayItem – plays a specific song number in the playlist
- FF – fast forward
- Rew – rewind
- Next – plays next song
- Previous – plays previous song
- IncreaseVol – increases the volume
- DecreaseVol – decreases the volume
- PlaylistAdd – adds a song to the playlist
- PlaylistDelete – deletes a song from the playlist
- PlaylistClear – clears the playlist
- PlaylistShuffle – shuffles the playlist

Music Database Device Resource The music database device holds information about what songs are available. Its only device action is the following:

- ReadMusicDB – scans a pre-specified directory for MP3 files and adds posts containing their full path, artist name and song title to the database.

The device queries are:

- **?*x*.path(*x*)** – find out the directory path of a song
- **?*x*.available_song(*x*)** – find out whether a certain song is available, given an artist name
- **?*x*.available_artist(*x*)** – find out whether a certain artist is available, given a song name

Dialogue examples

Example interaction 1 The following example dialogue (22) shows how a naive user can navigate through the menu structure using speech only.

(22) SYS> Hello.

SYS> Let's see. What can I do for you?

SYS> Do you want to control playback or manage the playlist? *(DynGUI displays two buttons, one with the text "control playback" and the other with the text "manage the playlist")*

USR> manage the playlist

SYS> Okay. Let's see. Do you want to add a song to the playlist, delete a song from the playlist, clear the playlist or shuffle the playlist? *(DynGUI displays four buttons, one for each alternative)*

USR> add a song to the playlist

SYS> Okay. Let's see. Which song do you want to add? *(DynGUI displays the text "Which song do you want to add?" and a text input field)*

USR> london calling by the clash

SYS> Okay. I have added a song to the playlist. *(The song London Calling by the Clash appears on the MP3 Gui playlist)*

Example interaction 2 The next example (23) shows how the user can accomplish the same thing using graphical input.

(23) SYS> What can I do for you?

SYS> Do you want to handle the player or manage the playlist? *(DynGUI displays two buttons, one with the text "control playback" and the other with the text "manage the playlist")*

USR> *(clicks on the"manage the playlist"-button)*

SYS> Okay. Let's see. Do you want to add a song to the playlist, delete a song from the playlist, clear the playlist or shuffle the playlist? *(DynGUI displays four buttons, one for each alternative)*

USR> *(User clicks on the"add a song to the playlist"-button)*

SYS> Okay. Let's see. Which song do you want to add? *(DynGUI displays the text "Which song do you want to add?" and a text input field)*

USR> *(User clicks on the song "Clash - Should I Stay Or Should I Go" in the MP3 GUI)*

SYS> Okay. I have added a song to the playlist. *(The song London Calling by the Clash appears on the MP3 Gui playlist)*

Example interaction 3 The next example (24) shows an example of a multimodal user utterance. It also shows an example of GODIS grounding. Since graphical input is assumed to be perfectly recognized, the system will only ask a check-question regarding the spoken part of the utterance:

(24) SYS> Hello.

USR> add this (low recognition score) *(clicks on "Ace Of Base - All That She Wants" in the MP3 Gui)*

SYS> Okay. You want to add a song to the playlist? *(DynGUI displays the text "You want to add a song to the playlist?", a "yes" button and a "no" button)*

USR> *(clicks on the DynGUI "yes" button)*

SYS> Okay. I have added a song to the playlist. *(The song All That She Wants by Ace Of Base appears on the MP3 Gui playlist)*

Example interaction 4 The following two dialogues are examples of dialogue with multiple applications (see section 2.1.2 and deliverable D2.2 ([19])). In addition to DJ-GODIS, the user here interacts with the GODIS-DELUX application described in section 2.3.6.

Example dialogue (25) shows an example of a multidomain utterance involving two independent tasks. Note that the tasks are specified in separate domain resource files which have no knowledge of each other. The two domains are glued together by a combined GF grammar, as described in section 2.2.5.

(25) USR> clear the playlist and turn on the light in the kitchen

SYS> Okay. I have cleared the playlist. *(The MP3 Gui playlist is cleared)*

SYS> Okay. I have turned on the light. *(The kitchen light is turned on)*

Example interaction 5 In example dialogue (26) the user switches to the GODIS-DELUX application and engages in a dialogue about turning on lights instead of answering a system question. When the light has been turned on, the system reraises the pending question of what song to add. Instead of answering the question the user asks the counter-question "What songs are available?". Since the artist issue has already been resolved for the Play-task, the system infers that the user wants to know about Madonna songs and answers the question. Finally the user selects one of the Madonna songs by clicking in the DynGUI.

(26) USR> play madonna

SYS> Okay. Let's see. Which song do you want to play? *(DynGUI displays the text "Which song do you want to play?" and a text input field)*

USR> turn on the light in the living room

SYS> Okay. Let's see. Which lamp do you mean? *(DynGUI displays the text "Which lamp do you mean?" and a text input field)*

USR> the ceiling lamp

SYS> I have turned on the light *(The living room ceiling light is turned on)*

SYS> So. Let's see. Which song do you want to play? *(DynGUI displays the text "Which song do you want to play?" and a text input field)*

USR> which songs are available

SYS> Like a Prayer. Lucky Star. Material Girl. *(DynGUI displays three buttons, one for each song name)*

USR> User clicks on the "material girl" button

SYS> Okay. *(Material Girl by Madonna shows up on the MP3 Gui playlist and starts playing)*

2.3.6 GODIS-DELUX

GODIS-DELUX is a GODIS application for the in-home domain. The application lets you control the lights in a house and ask about the status of a specific lamp (if it is on or off) or ask, in general, which lamps are on or off. A lamp can also have a dimmer attached and this means that you can also dim or turn up the light on that lamp.

Scenario

(27) *User is late for work, in a hurry, and just about to leave the house.*

U: Turn off all lights

All lights in the house are turned off

Later that night the user arrives home from work carrying bags in both hands.

U: Turn on the hall light and living room light

The hall light and living room light is turned on

User puts down his bags, walks into the living-room, sits down on the sofa and turns on the TV

U: Dim the lights

Living-room lights are dimmed

U: Is the hall light on?

S: Yes, it is on.

U: turn it off

Infrastructure

The application uses the GODIS dialogue manager and the TRINDIKIT4 dialogue system toolkit and consists of a collective of OAA agents organized as in figure 2.9. The Controller agent, DME agent and the MMD, ASR and TTS agents are TRINDIKIT agents, which communicate using the TRINDIKIT4 OAA API described in D5.1.2 [3]

- The controller agent coordinates the different modules and agents by executing a set of serial control algorithms in parallel.
- The timeout agent is used by the controller to determine when the user's turn is over.
- The DME agent holds the total information state (TIS) and the core dialogue management modules, update and select, as well as interpretation and generation modules.
- The actual interpretation and generation is done by the GF agent, which is called over OAA by the interpretation and generation modules.
- The DynGUI input/output module agent is used to dynamically render graphical menus which can be used for graphical input.
- The ASR module agent continuously listens for input and writes the recognized result to TIS.
- The TTS module agent reads output from TIS and synthesizes it as speech, when called from the controller.
- The GODIS-DELUX GUI agent displays a schematic map of the house showing each lamp in its specified location. According to the user actions the GUI is modified to reflect the current status of each lamp in the house.

Research issues addressed

Dynamic reconfiguration GODIS-DELUX demonstrates implicit application switching and offline plug-and-play, as described in D2.2 [19]. The DJ-GODIS and GODIS-DELUX applications have been implemented independently and no knowledge of the other application is hard-coded into grammars or resources. Nevertheless, they can be run simultaneously with a seamless interface presented to the user.

Integrated approach to multilinguality and multimodality By deploying multimodal (and multilingual) GF grammars, GODIS-DELUX demonstrates the integrated approach to multimodality and multilinguality described in Section 2.2 as well as in D1.2b [15] and D1.5 [14].

Multimodal menu-based dialogue GODIS-DELUX demonstrates the MMD approach to multimodal dialogue management, as described in Section 2.1.2 and D2.1 [18].

Asynchronous multimodal dialogue management GODIS-DELUX makes use of the asynchronous capabilities of TRINDIKIT4 as described in D5.1.2 [3] to enable multimodal barge-in and incremental interpretation.

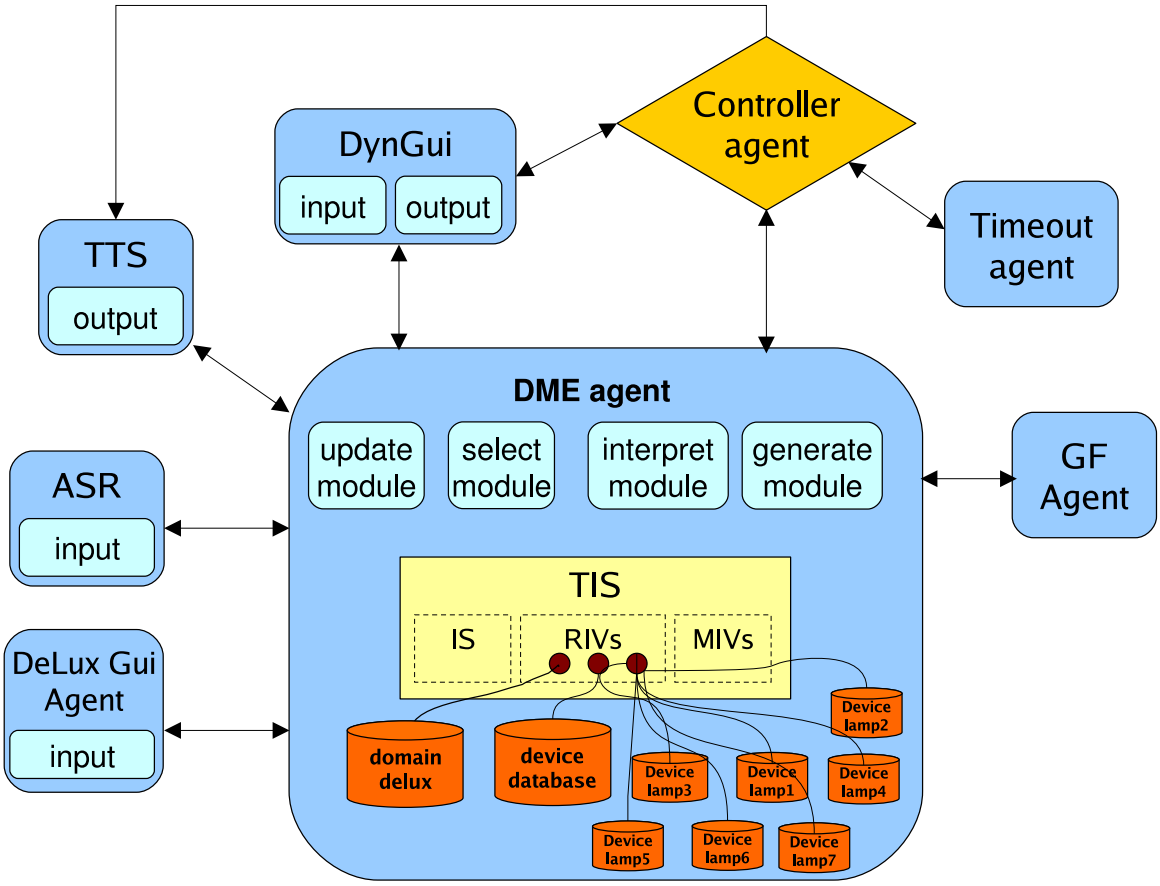

Figure 2.9: GODIS-DELUX system as a collective of agents

Functionality

The GODIS-DELUX application offers the following functionality. The user actions are:

- Turn on one or several lights. (”turn on all lights in the living room”)
- Turn off one or several lights. (”turn off the kitchen light”)
- Dim the light on one or several lamps. (”dim the living room light”)
- Turn up the light on one or several lamps. (”turn up the living room light”)

The issues that can be raised by the user are:

- Ask if a specific lamp is on. (”is the bedroom ceiling lamp on”)
- Ask if a specific lamp is off. (”is the kitchen light off”)
- Ask which lights are on. (”which lights are on in the bedroom”)
- Ask which lights are off. (”which lights are off”)

Multilinguality

GODIS-DELUX supports interaction in English and Swedish. GF grammars are used for both parsing and generation. The GF grammars for GODIS-DELUX are described in Section 2.2.6. The speech recognition grammars are automatically generated from GF. At any time during the dialogue the user can switch language using buttons in the the dynamic GUI (see Section 2.1.2).

Multimodality

GODIS-DELUX implements the MMD approach to multimodality described in Section 2.1.2, and uses the DynGUI (see Section 2.1.2) for graphical menu-based interaction. It also has an application-specific GUI.
The user can provide input to the application via either the GUIs or using speech. The output modalities are speech together with graphical output in the GUIs. The application is connected to the GUIs via an OAA agent. The GODIS-DELUX GUI graphically shows each lamp and its location in the house. The GODIS-DELUX GUI is shown in Figure 2.10.

Implementation of resources

Domain resource In GODIS-DELUX there are nine domain plans which are arranged in a very flat menu structure (with a depth of only two). There is one toplevel plan for selecting which subtask to engage in; four plans dealing with actions, and four plans dealing with issues. The action plans are:

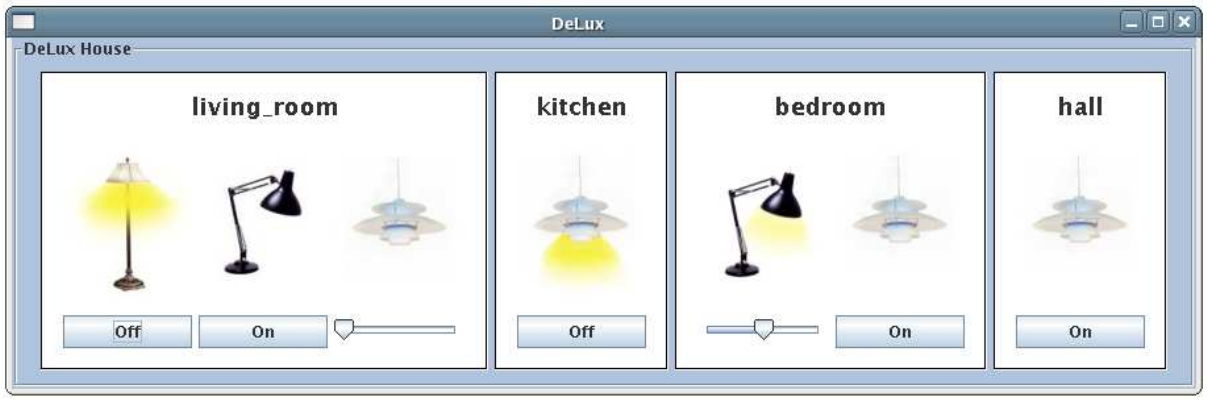

Figure 2.10: GODIS-DELUX GUI

- **turn_on_light**
- **turn_off_light**
- **dim_light**
- **undim_light**

The plans for dealing with issues are:

- **?***x***.light_on(***x***)**
- **?***x***.light_off(***x***)**
- **?light_status_on**
- **?light_status_off**

What the plans achieve is fairly easily understood from their names, and we will only include detailed descriptions of two of the plans, namely, the those corresponding to the action **turn_on_light** and the issue **?***x***.light_off(***x***)**.

In all action plans it is possible to specify several lamps or rooms by saying “all lamps” or “all rooms”, e.g., “turn on all lamps in the bedroom” or just “all lamps” as answer to the question “Which lamp do you want to turn on”.

First, we look at the plan corresponding to the action **turn_on_light**[8]:

[8]This dialogue plan has been slightly edited for easier comprehension.

ACTION : turn_on_light

PLAN:
- findout(**?***x*.**room_turn_on(***x***)**)
- findout(**?***x*.**lamp_turn_on(***x***)**)
- dev_queryAll(device_database,**?***x*.**device_turn_on(***x***)**)
- forall_dev_query(**?***x*.**light_off(***x***)**)
- forall_dev_query(**?***x*.**dimmer_on(***x***)**)
- if ¬**device(_)**
- then [report(turn_on_light,failed(¬∃*x*.**device(***x***)**))
- else
  - if ¬**light_off(_)** and ¬**dimmer_on(_)**
  - then [report(turn_on_light,failed(**all_lights_on**))
  - else [forall_dev_do(TurnOnLight)

POSTCOND : **done(TurnOnLight)** || **status(failed(_))**

Two findout actions is used to get the information regarding room and lamp position needed to perform the action. The dev_queryAll action queries the database device to find the relevant sockets for the devices which should be turned on, based on the specified room and lamp. If successful, the result of this query is the addition to the system's private beliefs of a set of propositions **device(***d***)** where *d* is a device socket.

Next, to check for possible problems and informing the user of these (e.g if the lights are already on, or if the specified lamp does not exist in the specified location) , the plan first queries each device fulfilling the given constraints on room and lamp position (forall_dev_query(**?***x*.**light_off(***x***)**) and forall_dev_query(**?***x*.**dimmer_on(***x***)**) for their status and then, using conditionals, an appropriate action is taken.

If no lamp fulfilling the given constraints exists, the system reports this, using:

report(turn_on_light,¬∃*x*.**device(***x***)**)

If all specified lights are already on it reports this (report(turn_on_light, **failed(all_lights_on)**)) otherwise it tells each lamp device fulfilling the constraints to perform the TurnOnLight action (forall_dev_do('TurnOnLight')).

The postconditions are used to check whether an action is considered done. In this case this is true either when the action has been performed or when the system has reported a reason for not performing the action (thus adding a proposition indicating **status** to the shared commitments).

The plan shown next corresponds to the issue **?***x*.**light_off(***x***)** which would be activated if the user e.g., asked the question "Which lamps are off in the living room"[9].

[9]This dialogue plan has been slightly edited for easier comprehension.

ISSUE : **?***x*.**light_off(***x***)**

PLAN:
- bind(**?***x*.**room_light_off(***x***)**)
- bind(**?***x*.**lamp_light_off(***x***)**)
- dev_queryAll(device_database,**?***x*.**device_light_off(***x***)**)
- if ¬**device(_)**
- then [assume(¬∃*x*.**device_light_off(***x***)**)
  - forall_dev_query(**?***x*.**light_off(***x***)**)
- else
  - if¬ light_off(_)
  - then [assume(¬∃*x*.**light_off(***x***)**)

First, two bind actions are used to get optional information from the user. If the questions **?***x*.**room_light_off(***x***)** or **?***x*.**lamp_light_off(***x***)** are addressed in the user utterance the answer will be integrated with the question and stored among the shared commitments. If not addressed, the two bind actions will be popped off the plan. As opposed to findout actions, bind actions are not actively performed by the system, i.e., they are not realized as **ask** moves.

Then, as for the plan **turn_on_light**, the dev_queryAll action queries the database device to find the relevant sockets, with possible restriction to a specified room or lamp.

Then, if there are no **device(_)** propositions in private beliefs (the database query for device sockets returned an empty answer), the system assumes that this is because there exists no such lamp; this will eventually be gives as the answer to the user.

If there are **device(_)** propositions among the private beliefs, the system queries all lamp devices to get their status. This information is stored as a set of propositions in the system's private beliefs. All propositions **light_off(***d***)** stored in private beliefs together with the unresolved issue **?***x*.**light_off(***x***)** will eventually trigger the system to perform an answer move. If none of the lamps are off, the system answers by saying this, otherwise, it answers by enumerating all lamps that are off.

Plans dealing with issues have no postconditions. Instead, issues are considered resolved when an answer to the raised issue exists in the shared commitments.

Device resource Each specific lamp in the house is connected to a socket, and each socket is controlled by a lamp device. In this implementation there are seven lamps and thus also seven lamp devices. Each lamp device holds information about the status (on/off, dimmed etc.) of the lamp.

Each lamp device can execute the following actions performed as OAA solvables:

- action TurnOnLight
- action TurnOffLight
- action DimLight
- action UndimLight

Each lamp device can also execute 7 different queries which are used to find out e.g., if the lamp is on, if the dimmer is on, if it is already dimmed as much as it can be etc. These queries are called from within plans to get the information needed to decide on an appropriate action to take.

The database device is mainly an interface to the database but also used by the GODIS Dialogue Move Engine to compute inference. An example from the GODIS-DELUX application would be if a user says "turn on the light in the kitchen". According to the database only one lamp exists in the kitchen and therefore the system can infer that this must be the lamp the user wants to turn on. Therefore the system can omit the question regarding which lamp to turn on and instead, without asking, turn on the lamp. Another example would be if the user made a request to dim the light in a room where only one lamp has a dimmer attached. Here, there is no need to ask the user to specify which lamp she wants to dim.

Database resource The specification of which lamps exist and their location in the house is made in a prolog database. Each lamp device is also connected to a socket. The database is also used as input to the GODIS-DELUX GUI agent.

Dialogue examples

We now give sample interactions with the GODIS-DELUX application.

Example interaction 1 In this interaction we see how the system uses inference to compute which lamp to turn on after the user has answered "kitchen" to the question "In which room do you mean" (only one lamp exists in the kitchen). Due to this it does not need to ask the question "Which lamp do you mean". This is also the case when the user makes a request to dim the light in the living room. The only lamp that has a dimmer attached is the ceiling lamp so the question regarding which lamp to turn on can be omitted.

(28)
S: What can I do for you?
U: turn on the light
S: Okay. Let's see. In which room do you mean?
U: the kitchen
S: Okay. I have turned on the light.
S: Returning to restart. What can i do for you?
U: turn on the livingroom light
S: Okay. Let's see. Which lamp do you mean?
U: the ceiling lamp
S: Okay. I have turned on the light.
S: Returning to restart. What can i do for you?
U: which ceiling lamps are on
S: The kitchen ceiling lamp. The livingroom ceiling lamp.

U: dim the livingroom light
S: Okay. I have dimmed the light.
U: turn off all lights in all rooms
S: Okay. I have turned off the light.

Example interaction 2 In this interaction we see how the system reports a reason for not performing the requested action. In this case it is because there is no floor lamp located in the kitchen. We also see how the system decides to give interrogative feedback to ground the user utterance "the desktop lamp" which has got a low recognition score (indicated by 0.3).

(29)
S: What can I do for you?
U: turn on the kitchen floor lamp
S: Okay. Let's see. I failed to turn on the light because there is no such lamp in the room
S: Returning to restart. What can i do for you?
U: turn on the bedroom light
S: Okay. Let's see. Which lamp do you mean?
U: the desktop lamp 0.3
S: You mean the desktop lamp, is that correct?
U: yes
S: Okay. I have turned on the light.
S: Returning to restart. What can i do for you?
U: dim the bedroom light
S: Okay. I have dimmed the light.
S: Returning to restart. What can i do for you?
U: is the hall light off
S: The light is off.

2.4 Conclusion

This chapter has addressed the issue of multimodality and multilinguality in GODIS, focusing in particular on a unified approach to multimodality and multilinguality using GF, and on the development of GODIS applications that offer practical solutions to theoretical issues of multimodality and multilinguality.

We have shown how dialogue management can be straightforwardly adapted in an information-state-based system to handle both multimodality and multilinguality. For GODIS this has involved extensions to and modifications of the information state, the update rules, and the overall system architecture and control. The highly modular architecture of GODIS has greatly facilitated the incorporation of new modalities and languages.

The technique of MMD (Multimodal Menu-Based Dialogue), whereby an existing graphical interface for some device is converted into dialogue plans in GODIS, has been shown to create useful and natural multimodal dialogue, allowing communication in both speech and graphics – separately or simultaneously – with GODIS offering its full range of flexible and advanced dialogue management capabilities also for multimodal interaction. MMD is incorporated in three of the GODIS applications showcased: DJ-GODIS, GOTGODIS, and GODIS-DELUX.

The ISU approach and the modularity of GODIS also give clear advantages for multilinguality, both at the development stage, where only clearly separated language-specific components need to be modified, and at run-time, when the separation of language-specific information from other parts of the system make it possible to change languages in the middle of a task, maintaining the information state as it were before the language change. As we have shown, no extensive modifications of dialogue management are needed for the inclusion of new languages in GODIS. We have provided two different ways of achieving language change in GODIS applications: using speech in the middle of a dialogue in AGENDATALK, and by clicking a check box in the DynGUI for DJ-GODIS, GOTGODIS, and GODIS-DELUX.

Grammar development using GF has made use of the separation of abstract syntax from concrete syntax, where an abstract syntax has been related to several different concrete syntaxes, each concrete syntax corresponding either to a language or a modality. The unification of multimodality and multilinguality in GODIS/GF has thus been approached in a very direct and explicit way, with the same underlying representations connecting both different languages and different modalities.

The grammar work has involved an application-independent multilingual and multimodal resource grammar containing all contributions in common for the applications. The rapid development of this grammar has been enabled through the already existing GF Resource Grammar Library, currently available in eleven languages, of which so far at most seven has been used for a GODIS application.

In addition to this common grammar, a number of different application-specific grammars have been implemented for all the applications, handling the particulars of each application. Detailed descriptions of the applications themselves, including their precise multimodal and multilingual behaviour, and the interactions they allow, give practical proof of the feasibility of the theoretical issues involved in the unification of multimodality and multilinguality in the in-home domain. Implementational specifications are further given in Appendix A, separated into the grammars that are developed, application by application, and all other relevant files for each application.

The integration of GF and the ISU-based GODIS has provided a highly workable and productive environment for rapid prototyping of dialogue applications for new domains and new languages, using a unified approach to multimodality and multilinguality. The unified approach provides a coherent, powerful, and flexible technique for interactive systems.

Chapter 3

Multimodality and Multilinguality in the Linguamatics Interaction Manager

3.1 Introduction

The primary focus of the Linguamatics Interaction Manager is domain reconfigurability. In this chapter we look at how multimodality is currently handled by the Linguamatics Interaction Manager, and explore the relationship between reconfigurability and multimodality and multilinguality, and how a uniform approach fits with this.

3.2 System Summary

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 use in dynamic scenarios where the whole task or ontological structure, and the vocabulary can change. This contrasts with more standard scenarios where the task and ontological structure remain constant, and the only change is in the instantiations e.g., the contents of a database for a flight booking database.

The home environment is a particularly good example of a dynamic scenario. There is no fixed set of rooms, or fixed set of devices. Over time, new devices will need to be added, and new applications or services. The system has been installed at the Advantica Test House in Loughborough. It was configured for the 8 rooms in the house, and linked to a task manager to control a number of home devices, including lights and blinds. Loughborough University published the results of a trial using the system which involved twenty eight participants. Users found the voice activation clear, and useful. 81% of interactions achieved the goal at the first attempt [6].

3.3 Issues Addressed

The Linguamatics Interaction Manager takes a very strong approach to reconfigurability, allowing new applications and devices to be added, even at run-time. The theoretical approach taken combines Ontology-Based Dialogue with the Information State Update Approach. The main focus of Linguamatics work is WP2, and the description of the Interaction Manager is in Deliverables D2.1 [18] and D2.2 [19]. In the next two sections we will focus on how multimodality and multilinguality are treated in the system.

3.4 Multilinguality

Although multilinguality was not a focus of our work, we have made steps towards making the Linguamatics Interaction Manager multilingual. Most of the English specific implementation has been removed, although there are remnants, for example in the generation of definite descriptions, both for output and for language models.

Two main approaches to multilinguality were considered. The first was to treat multilinguality as an instance of reconfigurability. To use a new language, a completely new ontology would need to be provided, with new synonyms and preferred terms for each concept. This approach would have required relatively little change to the system, other than the removal of the remnants of English-specific implementation. However this approach would have some limitations. Firstly, it does not allow any mixing of different languages, which could be useful, for example, in providing multiple labels for terms on the same graphical display. Secondly, it means that ontologies for different languages can get out of step, which could cause maintenance problems in the longer term.

The second approach, and the one which has been adopted, is to allow for preferred terms and synonyms to be provided for more than one language. This is similar to the approach taken by the W3C SKOS Core Vocabulary scheme for thesauri written in RDF [28], which provides an optional language tag based on ISO-639-2 for synonyms and preferred terms. There are some scenarios however, where even this seems too limiting. For example, we may want different preferred terms for different purposes even though the language is the same. For example, “Sony 10789” may be appropriate for the shopkeeper, but “Sony 32inch HDTV” may be more appropriate for the shopper. This indicates that we may want to further distinguish preferred terms according not only to language, but also for purpose.

3.5 Multimodality

The current approach to multimodal input in the Linguamatics Interaction Manager is relatively simple. For input, it assumes independent events via menu clicks or via a spoken command. Output is achieved either by speech synthesis, or by changing the graphical display, or most commonly, using both modalities.

There is a small amount of adaptation across modalities according to which of the modalities are available. For example, if speech synthesis is available, low priority messages such as “Pardon” are only provided in the spoken modality, and not also on the graphical display. There is also some adaptation of messages. For example, if both speech and graphics are available the system will use the question “please give your name”. If the only modality available is graphical, this is changed to “please click on your name”.

All the modality settings which are described below can be changed by sending a message to the system from another application. This may be useful if, for example, the user is moving around the house and sensors can detect that the user will not be able to see a screen from their new position so will need a full set of options to be enumerated by voice. Changes to settings can also be made by users themselves by using speech or by using a touch screen as part of a normal interaction. This is achieved by including a “Settings” subontology. For example, the leaf node arrived at by traversing “Settings” then “Prompts” then “Verbose” is associated with a command to change the prompt setting to “Verbose”.

3.6 Speech Recognition

The system dynamically generates appropriate language models for the speech recogniser according to the ontology and the current Information State. There are three settings: safe, default and expressive. The default setting is currently equivalent to the expressive setting.

The safe setting is used in noisy environments. Language models are restricted to only recognise one of the currently available options, or escape options such as “help” or “back to the top”. The expressive setting not only includes the currently available options as defined by the Information State, but also any concepts subsumed by the current options according to the ontology. This allow users to skip many levels of interaction, for example by saying “cinema booking” when presented with the top-level menu items. The expressive setting also includes a larger contextual grammar if this is available. For example, in the home domain there is a statistical language model which allows full commands such as “turn on the light in the living room”. The statistical language model uses dynamic classes, and these are populated according to the ontology and the current context (which is part of the Information State). For example, at the top level of the home, the language model includes devices which exist in this home according to the ontology. If the context moves to the kitchen, the language model only includes kitchen devices.

The current expressive setting therefore allows reference to the current options or anything more specific. This allows the vocabulary to be kept relatively small at all times, while coping with most utterances that are appropriate for the user to say in the given context. Utterances which are not currently captured are ones which would involve a jump out of context e.g., saying “incidentally can you check the cooker is off” when in the middle of a cinema booking dialogue.

3.7 Multimodal Output

Multimodal output combines spoken output with HTML or XML for graphical output. The set of options displayed on screen is generated dynamically from a combination of the Information State and the ontology. For example, when traversing a menu structure the alternatives shown will correspond to the visible concepts below the current concept. When asking for a clarification, the set of options will be created dynamically from the Information State. More information on the graphical output is provided in Deliverable D2.1 [18].

Spoken output has three settings: minimal, standard and verbose. The standard setting is currently equivalent to the minimal setting. The minimal setting tries to keep prompts as short as possible. This is particularly appropriate if the user has a view of the available options on screen. Options are enumerated if there are just two options. Otherwise, the system asks for a concept which covers all the items in the option list. For example, given the options, lounge television, hall light and dining room radio, the standard output will be "which device?". The verbose output is useful if, for example, the user has poor sight, or is not looking at the screen. This enumerates all the options currently available. For the example above, the output using the verbose setting would be "would you like lounge tv, hall light or dining room radio?". Prompts are synthesised to provide spoken output, and also rendered on the screen as text.

3.8 Moving to a unified approach to multimodality and multilinguality

In the current Linguamatics System, there is a separate data structure which associates concepts with images. We are looking to replace this treatment with one where icons are provided as preferred terms, just in a different language. To achieve this we intend to include "graphics" as one of the possible languages. Allowing for multiple icons with different properties (small or large icons for example) will require a secondary tag, and seems parallel to allowing for different linguistic preferred terms using a "purpose" tag.

Although we intend to provide a similar datastructure to contain different views of an object, whether graphical or linguistic, and for different languages, this does not mean that generation within each modality will be exactly parallel. For example, we would expect to retain the current behaviour where the availability of a graphical modality will set the prompts to the "minimal" setting, unless set otherwise by the user.

3.9 The Home Domain Showcase

As an example for in-home applications, the Linguamatics Interaction Manager was linked to home simulation software developed by the University of Loughborough and configured with an example initial ontology including 15 rooms, 20 different kinds of devices or sensor (over 100 individual devices), and several different services including restaurant booking, and recipe

reading. There is also the ability to add extra devices and services at run time.

The graphical display was developed as part of the UK DTI funded Service Aggregation Trial and is shown in Figure “TV Interface”. The graphics were designed to be readable on a standard television linked to a set-top box.

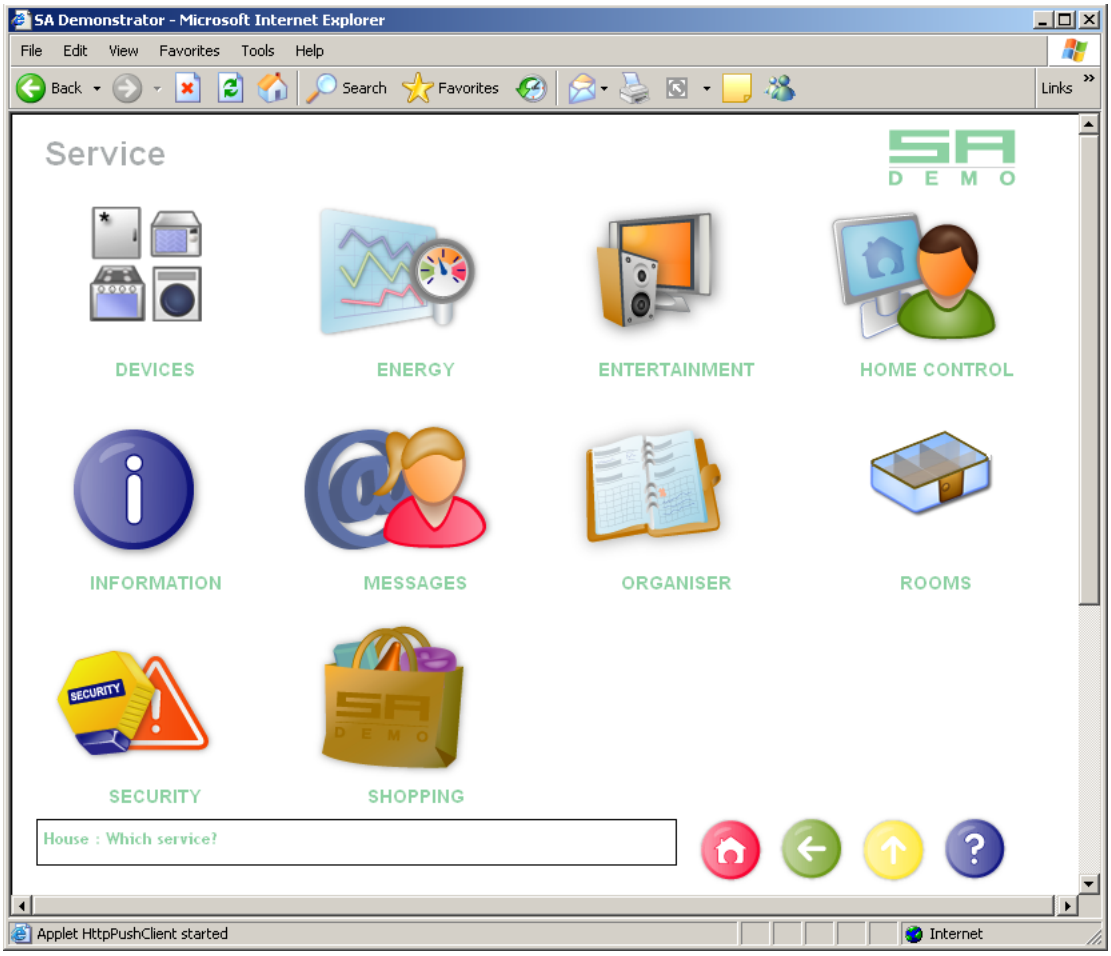

Figure 3.1: TV Interface

The graphical display provides a menu structure which the user can follow, either by clicking on options, or by uttering one of the options. However, users can also use voice to skip several stages. For example, “lounge please” would take the user to the lounge context. The graphics are then renewed to reflect all the devices in the lounge which could be controlled. A new prompt “which device?” is displayed and synthesised (assuming default prompt mode). The user can also take initiative and provide a full command e.g., “Turn the tv to channel four”. The system will attempt to execute this command, and will respond according to the state of the house as returned by the home simulator, with either: “turned the tv to channel four” or “the tv is already on channel four”. If a command is not fully specified, the system will take back initiative and ask for clarification e.g., asking “which television?” if there is more than one in the current context.

3.10 Conclusion

In this chapter we have described how multimodality is currently handled in the Linguamatics system, and how the language models and the output are dependent on the ontology and the Information State. We have also outlined an approach which treats multilinguality similar to multimodality, and how this would fit with an ontology-based framework. The combination of Information State for providing a description of context, and the use of a single ontology to describe the domain knowledge has provided a powerful framework where the input (the language models for the speech recogniser) and the set of multimodal responses are both relatively simple functions from the Information State, the ontology and the modality settings.

Chapter 4

Multimodality and Multilinguality in MIMUS

4.1 Introduction

Task 1.6 in TALK involved a *Proof of concept dialogue system using the multimodal grammar library*. This chapter summarizes MIMUS, the showcase of a unified approach to multimodality and multilinguality in the In–Home domain. Although the scenario has already been described in previous reports, a brief description has nonetheless been included. The chapter continues then to summarize the WoZ experiments results and their impact on the final showcase. The following section provides an overview of the infrastructure, which outlines the main MIMUS agents and their interconnection. Then, the main research topics addressed by USE during the project are described, specially multimodality and multilinguality. Finally, a complete set of dialogue examples is presented, as well as a sum up with the main conclusions about MIMUS.

4.2 Scenario

As mentioned in previous Deliverables, USE has worked on the development of multimodal and multilingual applications in the In–Home domain. Although the results can be extrapolated to other user profiles, MIMUS has focused on wheel–chair bound users and their special circumstances.

In this particular scenario, users are able to access the system from their wheel–chair through different modalities, that is, using speech and/or a graphical interface (see Figure 4.1). The scenario includes microphones, speakers and a touch screen where the information can be displayed and introduced or selected.

Wheel–chair bound users represent a particularly interesting set of users due to their difficulties to move around and therefore their motivation to use the system, and due too to the wheel–chair itself and the assumption that the touch–screen would be available to them at all times. In this particular case, enquiring about the home and the status of all devices becomes more than ever

Figure 4.1: Scenario Description

an important issue.
MIMUS lets the user control On–Off devices and dimmers (with values ranging between “0” and “100”, e.g., blinds). In this context, “control” means both changing their state or enquiring about it. The user can interact with the system naturally using speech or the tablet–PC pen. A set of dialogue examples illustrating MIMUS interaction capabilities is presented in section 4.6.

4.2.1 WoZ Experiments

In order to collect first–hand information about the users’ natural behavior in this scenario, USE has conducted several WoZ experiments. A rather sophisticated multilingual WOZ experimental platform was built for this purpose. This platform however has ended up being also an interesting result of the project, since it may be used for future research in other experiments.

The set of WOZ experiments conducted at USE was designed in order to collect data. In turn, this data helped determine the relevant factors to configure multimodal dialogue systems in general, and MIMUS in particular. Additional relevant information was also collected:

- any possible obstacles or difficulties to communicate
- any biases that prevent the interactions from being completely natural
- a corpus of natural language in the home domain
- modality of preference in relation to task
- modality of preference in relation to task and scenario
- output modality of preference in relation to the type of information provided
- modality preference in relation to system familiarity
- task completion time

- combination of modalities for one particular task
- inter–modality timing
- user evolution, learnability and change in attitude
- how additional modalities affect interaction in other modalities
- context relevance and interpretation in multimodal environments
- pro–activity and response thresholds in multimodal environments
- relevance of scenario specific–factors/needs
- multimodal multitasking: multimodal input fusion and ambiguity resolution

A detailed description of the results obtained after the analysis of the experiments and their impact on the overall design of the system may be found in Deliverable D6.4 Part II.

4.3 Infrastructure

MIMUS has been developed as a set of OAA agents linked through the central OAA Facilitator. Most of the agents are either clients or servers, but some of them have a double role, providing and using solvables from other agents. An overall view of the system is provided in figure 4.2:

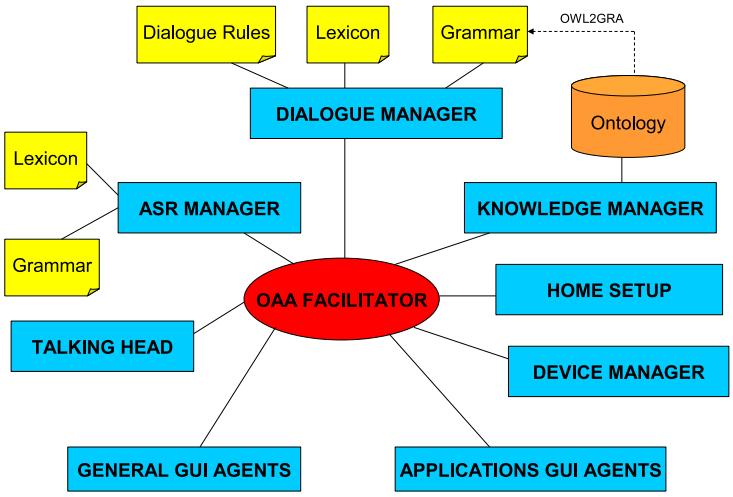

Figure 4.2: Architecture

The main agents in MIMUS are briefly described hereafter:

- The system core is the **Dialogue Manager**, which processes the information coming from the different input modalities agents and provides the appropriate output. It is by means of the output modalities agents that the DM can do this, while taking into account the contextual information in the Ontology and the application agents.
- The main input modality agent is the **ASR Manager**. MIMUS can work with any ASR as long as there is an OAA wrapper with the solvables described in [2]. USE has implemented these wrappers for Nuance and for Atlas.
- The **HomeSetup** agent displays the house layout, with all the devices and their state. All the information about the house elements (including walls, lamps, blinds, etc.) is loaded from the common knowledge resource: an OWL ontology. Whenever a device changes its state (i.e., a light is switched on), the HomeSetup is notified and the graphical layout is updated.
- The **Device Manager** controls the physical devices. The current implementation uses the X10 protocol. When a command is sent, the Device Manager notifies the HomeSetup and the Knowledge Manager, guaranteeing the coherence of all the elements in MIMUS.
- The **Knowledge Manager** is a key part of MIMUS, connecting all the agents to the common knowledge resource, by means of an OWL Ontology.
- The **Talking Head** is a new feature in MIMUS and represents a significant improvement with respect to the previous wrapper for Microsoft Animated Agents. MIMUS virtual character is also known as Ambrosio, and includes complex phoneme–viseme synchronization strategies (Loquendo TTS), and the ability to express emotions and play some animations such as nodding or shaking the head. A more detailed description can be found in [27].

A detailed agent description can be found in [2], including the solvables offered by each agent. A sequence diagram for a simple interaction where the user asks (verbally) to switch on the kitchen light is presented in Figure 4.3.

In this figure the Dialogue Manager receives the verbal input (“switch on the kitchen light”); this is parsed by its NLU submodule, yielding the following Information State [1]:

$$
\begin{bmatrix}
DMOVE & : & specifyCommand & \\
TYPE & : & On & \\
ARGS & : & [DeviceType, Location] & \\
 & & \begin{bmatrix} DMOVE & : & specifyParameter \\ TYPE & : & DeviceType \\ CONT & : & light \end{bmatrix} & \\
DeviceType & : & & \\
 & & \begin{bmatrix} DMOVE & : & specifyParameter \\ TYPE & : & Location \\ CONT & : & kitchen \end{bmatrix} & \\
Location & : & &
\end{bmatrix}
$$

[1] The Information State has been deliberately simplified for clarification purposes. The multimodal metadata, for instance, will be described in section 4.4.2

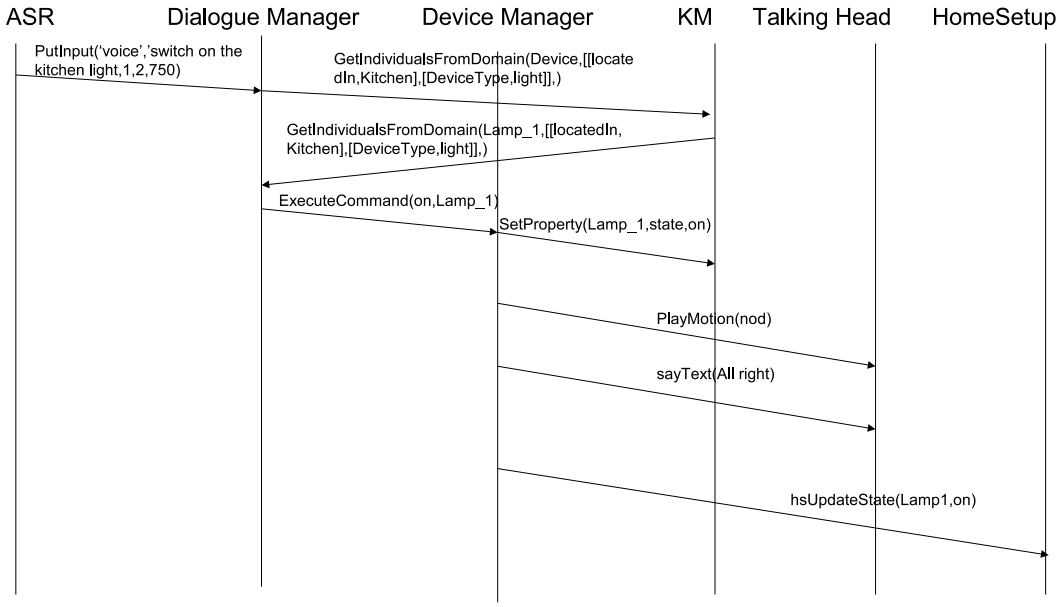

Figure 4.3: Sequence Diagram

The Dialogue Manager then performs reference resolution, searching the ontology (by means of the Knowledge Manager agent) for all individuals whose "DeviceType" is "light" and whose "Location" is "kitchen". The Dialogue Manager builds this query automatically from the "Type" and "Cont" fields, keeping it therefore domain independent. The answer from the Dialogue Manager (there is one light in the kitchen) is used to update the Information State:

$$\left[\begin{array}{lcl}
DMOVE & : & specifyCommand \\
TYPE & : & On \\
ARGS & : & [DeviceType, Location] \\
 & & \left[\begin{array}{lcl} DMOVE & : & specifyParameter \\ TYPE & : & DeviceType \\ CONT & : & light \end{array}\right] \\
DeviceType & : & \\
 & & \left[\begin{array}{lcl} DMOVE & : & specifyParameter \\ TYPE & : & Location \\ CONT & : & kitchen \end{array}\right] \\
Location & : & \\
 & & \left[\begin{array}{lcl} Quantity & : & 1 \\ & & \left[\begin{array}{lcl} Label & : & Lamp_1 \\ Location & : & Kitchen \\ DeviceType & : & Light \end{array}\right] \end{array}\right] \\
ReferenceResolution & : & RR1 \quad :
\end{array}\right]$$

This updated Information State is now used by the Dialogue Manager to send the correct command to the Device Manager Agent. The latter translates the command to the appropriate X10 physical instruction (to switch on the actual device), updates the ontology through the Knowledge Manager and updates the House Layout through the Home Setup agent.

Finally, the Dialogue Manager commands the talking head to confirm the correct execution of the command. This confirmation could be verbal, visual or both.

4.4 The ISU Approach in MIMUS

The main element of the ISU approach in MIMUS is the dialogue history, represented formally as a list of dialogue states. Dialogue rules update this structure either by producing new dialogue states or by supplying arguments to existing ones.

4.4.1 DTAC Information States

The information state in MIMUS is represented as a feature structure (also called a DTAC structure) which originally (i.e., before the TALK project) had four attributes: **D**ialogue **M**ove, **T**ype, **A**rguments and **C**ontents. A detailed explanation of the meaning of these features can be found in the Siridus Deliverable D3.2 [22].

1. **DMOVE**: This feature identifies the kind of dialogue move.

2. **TYPE**: This feature identifies the specific dialogue move in the kind of the corresponding DMOVE. While the DMOVE classification intends to be domain and implementation independent, the set of TYPEs will be domain dependent. In some sense, the TYPE classification instantiates the DMOVE model to the specific domain.

3. **ARGS**: The ARGS feature specifies the argument structure of the DMOVE/TYPE pair.

The following example illustrates a DTAC representation for the utterance *Llama a luis (Call luis)*.

$$\begin{bmatrix} DMOVE & : & specifyCommand & \\ TYPE & : & MakeCall & \\ ARGS & : & [Dest] & \\ & & \begin{bmatrix} DMOVE & : & specifyParameter \end{bmatrix} & \\ Dest & : & \begin{bmatrix} TYPE & : & Name \end{bmatrix} & \\ & & \begin{bmatrix} CONT & : & luis \end{bmatrix} & \end{bmatrix}$$

More attributes may be added in the course of the dialogue update, as for example the expectations (EXPT) generated by each dialogue rule. As illustrated in forthcoming sections, during the TALK project, additional attributes were added to the original four: Modality, Initial Time and End Time. The information state may be updated by a certain set of **update rules**, which may in turn be triggered by a specific set of **dialogue moves**. The latter contain declarative information, specific instructions to update the information state. Each rule consists of a rule name, a priority level, preconditions (*TriggeringConditions*) and actions (*PreActions*, *PostActions*, and *RecoveryActions*). Additional **update strategies** determine the specific rule(s), from the set of applicable ones, that must be used at any given time.

4.4.2 Multimodal DTAC structure

As illustrated in the previous section, the original DTAC structure is no longer sufficient. Modality and Time information are needed in order to implement fusion strategies at dialogue level. The new extended DTAC has therefore three new attribute–value pairs:

- MODALITY
- TIME_INIT
- TIME_END

This additional information is also useful in terms of presentation strategies (multimodal output), since the input modality is one of the relevant factors to determine the output modality/ies.

These new fields (together with the ASR confidence score) are considered to be meta–information because they are not semantic or syntactic constituents of the user's utterance. Therefore, it seems convenient to group them under a "meta–info" special attribute.

This new multimodal DTAC structure is illustrated below:

$$\begin{bmatrix} DMOVE & : & specifyCommand & \\ TYPE & : & MakeCall & \\ ARGS & : & [Dest] & \\ Dest & : & \begin{bmatrix} DMOVE & : & specifyParameter \\ TYPE & : & Name \\ CONT & : & luis \end{bmatrix} & \\ META-INFO & : & \begin{bmatrix} MODALITY & : & VOICE \\ TIME\_INIT & & 00:00:00 \\ TIME\_END & & 00:00:00 \\ CONFIDENCE & & 700 \end{bmatrix} & \end{bmatrix}$$

4.4.3 Updating the Information State in MIMUS

In this subsection we provide an example of how the Information State Update approach is approached in MIMUS. The MIMUS Dialogue Manager follows the dialogue rules manually defined by the designer. These dialogue rules are triggered by dialogue moves (any dialogue move whose DTAC structure unifies with the Attribute–Value pairs defined in the field TriggeringCondition) and may require additional information, defined as dialogue expectations (again, those dialogue moves whose DTAC structure unify with the Attribute–Value pairs defined in the field DeclareExpectations).

For instance, consider the following DTAC, which represents the information state returned by the NLU module for the sentence *switch on*:

$$\begin{bmatrix} DMOVE & : & specifyCommand & \\ TYPE & : & SwitchOn & \\ ARGS & : & [Location, DeviceType] & \\ META-INFO & : & \begin{bmatrix} MODALITY & : & VOICE \\ TIME\_INIT & & 00:00:00 \\ TIME\_END & & 00:00:00 \\ CONFIDENCE & & 700 \end{bmatrix} & \end{bmatrix}$$

Consider now a dialogue rule "**ON**" defined as follows:

```
( RuleID:    ON;    /* Rule name. */
  TriggeringCondition:
    (DMOVE:specifyCommand,TYPE:SwitchOn);  /* DMove that triggers this rule */
  DeclareExpectations: {
    Location<=(DMOVE:specifyParameter,TYPE:Location);
```

```
    DeviceType<=(DMOVE:specifyParameter,TYPE:DeviceType);
                                              /* Expectations linked to
                                                 the previous Dialogue Move.
                                                 The rule will not apply the
                                                 PostActions until the
                                                 expectations are fulfilled */
    }
  ActionsExpectations: {                    /* Actions to be executed when
                                                 an expectation is missing */
    [Location, DeviceType] =>
        {NLG(RequestLocationDeviceType);} /* What do you want to switch on? */
    }
  PostActions: {
                                              /* Actions to be executed
                                              when all the expectations
                                              are fulfilled */
     ExecuteAction(@is-ON); /* Sends the command to the Device Manager Agent*/
    }
)
```

The DTAC obtained for *switch on* triggers the dialogue rule **ON**, since that information state unifies with its TriggeringConditions. However, since two declared expectations are still missing according to this dialogue rule (**Location** and **DeviceType**), the dialogue manager will activate the ActionExpectations and wait for new inputs from the user.

Figure 4.4 shows a graphical sequence of how the Information State is updated when an expectation is fulfilled as a continuation of the dialogue above.

4.5 Multimodality and Multilinguality in MIMUS

This section describes USE's unified approach to Multimodality and Multilinguality, and how it has been implemented in MIMUS. Our approach is based on the combined use of two components in our system: the NLU module and the OWL Ontology. These modules are complemented with the ISU approach to dialogue management, therefore ensuring rapid porting to new domains and languages while keeping the naturalness and flexibility achieved through the ISU approach.

4.5.1 Integrating OWL in MIMUS

Initially, OWL Ontologies were integrated in MIMUS in order to improve its knowledge management module. This functionality implied the implementation of a new OAA wrapper capable of querying OWL ontologies.

As the project progressed, ontology–based dialogue management gained importance in MIMUS. Overall, the system is now much more coherent because multimodal and multilingual grammars

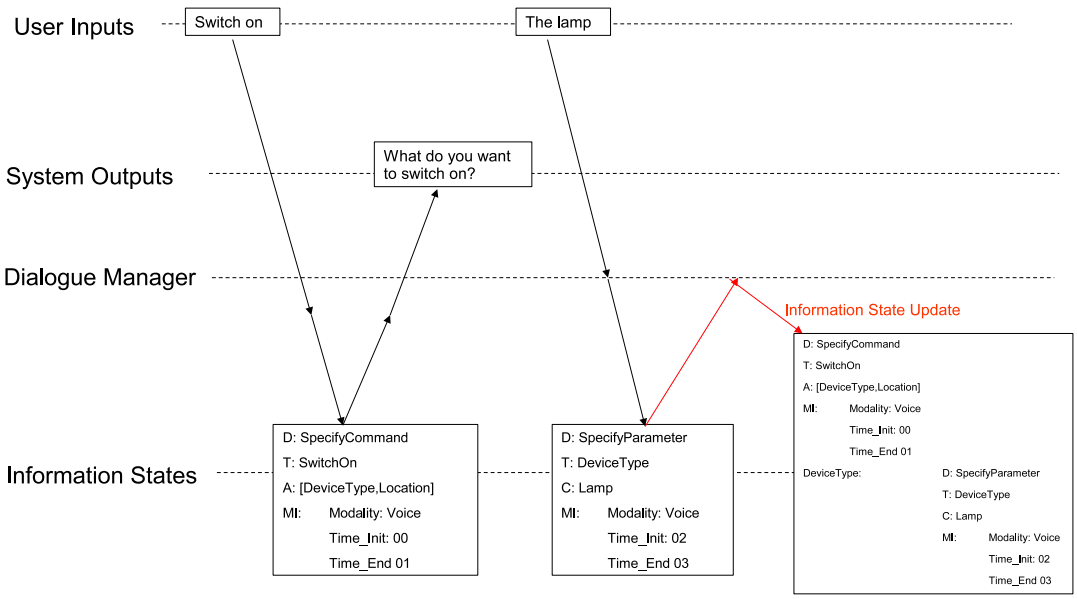

Figure 4.4: Information State Update

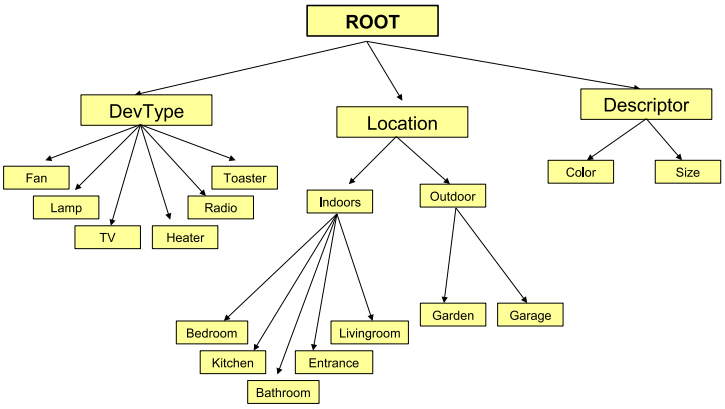

Figure 4.5: Former USE Ontology

can now be generated from OWL ontologies, and also, because the House layout is loaded from the OWL ontologies.

In former versions of the system a naïve ontology manager was implemented [21], which allowed us to define semantic graphs of the in–home domain. These graphs looked like the tree shown in figure 4.5.

In order to work with such graphs, a Knowledge Manager (KM) was built as an OAA agent that solved reference resolution queries over tree–shaped graphs. The goal of each query was to identify one or several unknown devices by means of a list of “positive” and “negative” filters. That is, it specified the values to be satisfied (or not) by the devices within the graph (color, device type, etc.). For instance, given the user request:

User Could you please turn on all the big lamps except the one in the bedroom?

the KM would solve a query such as:

```
KMDevRes(Positive[size:big,devtype:lamp],
Negative[location:bedroom])
```

As shown in this example, the KM enables the system to have a flexible interaction with the user using natural language commands and reference resolution. Nevertheless, this agent could be improved in a number of ways.

As the current needs were analyzed and some research on current standards and tools was carried out, it was determined that the already existing standard OWL in conjunction with the querying language RDQL and the reasoners included within the open–source Jena platform could help a

great deal, as detailed in Deliverable D2.1 [18].

Once it was determined that using OWL, Jena and RDQL the intended objectives could be achieved, an independent agent had to be implemented to substitute the previous Knowledge Manager. This agent had to comply with the following prerequisites:

Reusability: It should provide a mechanism to make abstract (domain independent) queries, useful for any ontology.

Expressivity: The agent should allow the dialogue manager to generate queries that might turn out to be necessary during the dialogue.

Both prerequisites were fulfilled by implementing two different OAA *solvables*:

- One to solve queries about the *domain* of the property.
- A second one to solve queries about its *range*.

This approach is completely independent of the particular ontology used and therefore reusable in any domain.

4.5.2 From OWL to the House Layout

MIMUS home layout does not consist of a pre–defined static structure only usable for demonstration purposes. It is actually dynamically loaded at execution time from the OWL ontology where all the domain knowledge is stored, assuring the coherence of the layout with the rest of the system.

This is achieved by means of the previously described OWL–RDQL wrapper. The Home Setup agent (please refer to D3.3 for a complete description) enquires through this agent the location of the walls, the label of the rooms, the location and type of devices per room and so forth, building the graphical image from these data.

4.5.3 From Ontologies to Grammars: OWL2Gra

As can be inferred from the previous sections, OWL ontologies play a central role in MIMUS. This role is limited, though, to the input side of the system. That is, the ontology is not used (yet) to generate dialogue rules, and is not used on the output side of the system, although the architecture designed for Multimodal Presentation assumes that OWL ontologies will also be used at this stage, as described in Deliverables D3.2 [9] and D3.3 [8].

In MIMUS, the domain–dependent part of multimodal and multilingual production rules for context–free grammars is semi–automatically generated from an OWL ontology. This approach is analogous to the combined use of GF and ontologies in Godis, although with less expressive power.

In MIMUS, this approach has achieved several goals: it leverages the manual work of the linguist, and ensures coherence and completeness between the Domain Knowledge (Knowledge Manager Module) and the Linguistic Knowledge (Natural Language Understanding Module) in the application.

Solution overview

The generation of linguistic knowledge from ontologies has been proposed previously. Russ et al. [26] proposed a method for generating context–free grammar rules from JFACC ontologies. Their approach was based on including annotations all along the ontology indicating how to generate each rule. They implemented a program that was able to parse the ontology and produce the grammar rules.

As previously mentioned, the USE approach focuses on grammar rules generation: no automatic lexicon hierarchy generation has been considered. To ensure coherence between the lexicon and the grammar, the list of potential non–terminal types is extracted from the list of all the entities within the ontology. The linguist decides which entities from this list shall remain in the final dialogue application.

It is worth noting that this tool is only meant as a helping device, a tool for the linguist. Therefore it does not provide a ready–to–use grammar. Using this tool, the grammar will be more easily generated and more consistent with the domain knowledge, but, in any case, the resulting grammar must be supervised and completed manually in a second phase.

Configuration files

As outlined above, the linguist must define a configuration file that will be used in conjunction with the ontology in order to generate the grammar rules. In this configuration file, the linguist has to identify the properties that may appear in the grammar and the way in which their domain and range will be included in the associated rules. In order to do it, an easy XML syntax has been defined (see DTD below).

Basically, the linguist can define the generation rules by means of nested *forEach* loops handling the properties (and subproperties) of the ontology, and using variables to identify the elements from its domain and range.

```
<!DOCTYPE rulesList [
  <!ELEMENT rulesList (forEach+)>
  <!ELEMENT forEach (forEach|rule+)>
  <!ELEMENT rule (left,right)>
  <!ELEMENT left (#PCDATA)>
  <!ELEMENT right (#PCDATA)>

  <!ATTLIST forEach property CDATA #IMPLIED>
```

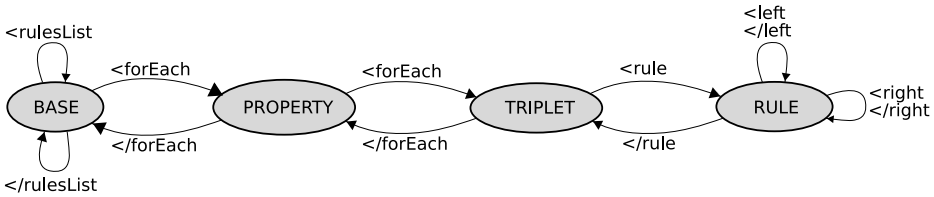

Figure 4.6: FSM for the configuration file parser

```
  <!ATTLIST forEach subPropertyOf CDATA #IMPLIED>
  <!ATTLIST forEach domain CDATA #IMPLIED>
  <!ATTLIST forEach range CDATA #IMPLIED>

  <!ATTLIST rule lang (ES|EN|GR) #REQUIRED>
]>
```

In order to better understand this structure as well as the objective of the tool, a set of examples including the relevance of the ontology, the configuration file and the resulting grammar rules are shown in the following sections.

Overview of the algorithm

In order to better illustrate how the algorithm works, this section will describe in more detail its functions. The algorithm consists of three major steps:

1. Parse the OWL ontology. The goal of this parsing is to generate an internal representation of the relevant ontological elements. This representation will in turn be used to make queries over the ontology.

2. Parse the configuration file. The objective here is to generate the list of all applicable rules.

3. Generate the output of rules. In this step, the script goes through the previous list of applicable rules, substituting the reference to classes and properties by the corresponding Input Form from the ontology.

The first two steps described have been implemented through a finite state machine (FSM) illustrated in Figure 4.6.
For each state in the FSM, only one set of attributes can be parsed. These are mentioned in the previous DTD structure:

Base :

- No attributes are expected in this state.

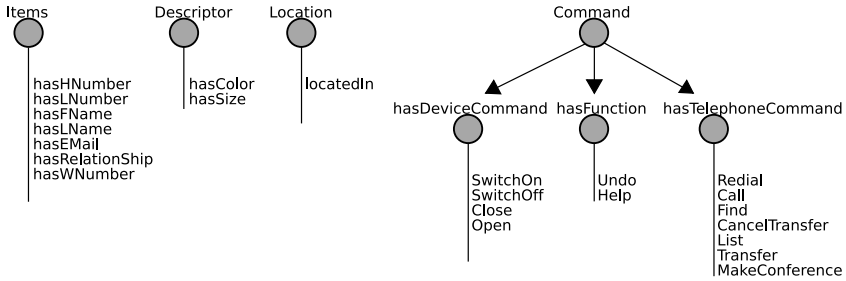

Figure 4.7: Ontology Structure

Property :

- propertyRef: Indicates the word that references the property in the rule description
- subPropertyOf: Indicates a super property. All the sub properties of this one (including the indicated one) will be treated by the algorithm

Triplet :

- domainRef: Indicates the word that references the domain in the rule description.
- rangeRef: Indicates the word that references the range in the rule description.

Rule :

- lang: Indicates what language the rule is valid for.

Sample Rules

The example below illustrates a common case in which the grammar rules will be generated. Our examples are taken from a smart–house domain in which the ontology describes both the hierarchy of devices in the house as well as the actions (or commands) which can be performed over those devices, such as *switch on the lamp in the kitchen.* Thus, consider an ontology where a set of properties are grouped as subproperties of a general **hasDeviceCommand** property. These properties are graphically displayed in Figure 4.7:

In this example we are going to analyze the portion describing the device–related commands. For example, the fact that the properties SwitchOff and SwitchOn have the class System as their domain and range over the classes Fan, Heater, Lamp, Radio and TV, is expressed in XML as follows:

```
<!-- hasDeviceCommand Subproperties -$\rightarrow$

 <owl:ObjectProperty rdf:ID="SwitchOff">
  <rdfs:subPropertyOf
    rdf:resource="#hasDeviceCommand"/>
    <rdfs:domain rdf:resource="#System"/>
```

```
 <rdfs:range>
  <owl:Class>
     <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Fan"/>
        <owl:Class rdf:about="#Heater"/>
        <owl:Class rdf:about="#Lamp"/>
        <owl:Class rdf:about="#Radio"/>
        <owl:Class rdf:about="#TV"/>
     </owl:unionOf>
  </owl:Class>
 </rdfs:range>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="SwitchOn">
    <rdfs:subPropertyOf
        rdf:resource="#hasDeviceCommand"/>
    <rdfs:domain rdf:resource="#System"/>
    <rdfs:range>
        <owl:Class>
          <owl:unionOf
            rdf:parseType="Collection">
            <owl:Class rdf:about="#Fan"/>
            <owl:Class rdf:about="#Heater"/>
            <owl:Class rdf:about="#Lamp"/>
            <owl:Class rdf:about="#Radio"/>
            <owl:Class rdf:about="#TV"/>
           </owl:unionOf>
        </owl:Class>
    </rdfs:range>
</owl:ObjectProperty>
```

Similarly, the properties Close and Open have System as their domain and Blind as their range.

```
<owl:ObjectProperty rdf:ID="Close">
    <rdfs:subPropertyOf
        rdf:resource="#hasDeviceCommand"/>
    <rdfs:domain rdf:resource="#System"/>
    <rdfs:range rdf:resource="#Blind"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="Open">
    <rdfs:subPropertyOf
        rdf:resource="#hasDeviceCommand"/>
```

```
        <rdfs:domain rdf:resource="#System"/>
        <rdfs:range rdf:resource="#Blind"/>
    </owl:ObjectProperty>
```

In this particular case, the linguist has detected that all properties are actually actions (expressed as the *Z* portion of the production below), that is, they correspond to the *Commands* to be performed by the system over all the elements in the range (expressed as the *Y* portion of the production below), in this case, all devices within the ontology. This can be easily expressed by the following configuration file, which will create rules of the form:

Command → Action_Property Device

```
<rulesList>
  <forEach property="Z" subPropertyOf="hasDeviceCommand">
    <forEach domain="X" range="Y">
      <rule lang="ES">
        <left>Command</left>
        <right>Z Y</right>
      </rule>
    </forEach>
  </forEach>
</rulesList>
```

Now, once the application is run indicating the appropriate configuration file, the following results are obtained:

Command → SwitchOff Fan
Command → SwitchOff Heater
Command → SwitchOff Lamp
Command → SwitchOff DimmerLamp
Command → SwitchOff Radio
Command → SwitchOff TV
Command → SwitchOn Fan
Command → SwitchOn Heater
Command → SwitchOn Lamp
Command → SwitchOn DimmerLamp
Command → SwitchOn Radio
Command → SwitchOn TV
Command → Close Blind
Command → Open Blind

It is important to note that even with this toy ontology, sixteen grammar rules have been generated using just two nested *forEach* loops.

A more detailed description of the strategies to generate multimodal and multilingual grammars from existing abstract knowledge representations in OWL is presented in Deliverable 1.5. [14].

4.5.4 Multilinguality in MIMUS

Multilinguality in MIMUS will be considered from two points of view. First, we will show how (some portion of) multilingual grammars can be generated through OWL2Gra. Then, we will outline how language change is achieved in the system.

Although there are no language restrictions, at the moment MIMUS is ready to be used in three languages: Spanish, English and German. USE has built full lexicon, grammar and dialogue specification modules allowing interactions in each of these three languages:

- The lexicon modules are obviously language–dependent.
- The grammar modules in MIMUS are semantically oriented as opposed to syntactically oriented grammars. These semantic grammars are automatically produced by OWL2Gra. However, after the automatic process, a manual review has proven to be mandatory.
- At dialogue level however and although the system may very well have separate modules, the configuration was simplified and the dialogue flow is identical for all three languages. This may not be generalized for any language under any domain, but works fine for restricted domains and similar languages such as these. As a result of this simplification, the dialogue specification module is currently shared by the three languages. It is important to note that even though this is so now, the system is not restricted in any way and it may have different dialogue configurations for each language. It is also important to highlight that MIMUS allows for language switching on–the–fly while keeping the dialogue context.

Capturing Multilinguality through OWL2Gra

Due to the structural differences among the human languages, different rules must be generated for different languages.

For example, to indicate the location of a given device, it would be *the kitchen light* in English, whereas in Spanish the word order changes: *la luz de la cocina* (the light of the kitchen).

Once the target language has been chosen, specific language rules must be generated.

Consider then the following fragment taken from the ontology previously shown, describing which elements can be affected by the property **locatedIn**:

```
<owl:ObjectProperty rdf:ID="locatedIn">
    <rdfs:domain>
        <owl:Class>
            <owl:unionOf
                rdf:parseType="Collection">
                <owl:Class rdf:about="#Lamp"/>
```

```
                <owl:Class rdf:about="#Radio"/>
                <owl:Class rdf:about="#Heater"/>
              </owl:unionOf>
          </owl:Class>
      </rdfs:domain>
      <rdfs:range>
         <owl:Class>
          <owl:unionOf
           rdf:parseType="Collection">
             <owl:Class rdf:about="#Bedroom"/>
             <owl:Class rdf:about="#Kitchen"/>
             <owl:Class rdf:about="#Hall"/>
             <owl:Class rdf:about="#LivingRoom"/>
          </owl:unionOf>
         </owl:Class>
       </rdfs:range>
</owl:ObjectProperty>
```

The multilingual configuration file that captures the structural differences mentioned above would be the following.

```
<rulesList>
   <forEach property="P"
    subPropertyOf="Location">
      <forEach domain="X" range="Y">
         <rule lang="ES">
            <left>X</left>
            <right>X P Y</right>
         </rule>
        <rule lang="EN">
            <left>X</left>
            <right>Y X</right>
        </rule>
      </forEach>
   </forEach>
</rulesList>
```

Now, if only English grammar rules are to be generated, the application must be run with the option ”-lang=EN”, obtaining the following result:

Lamp → Bedroom Lamp
Lamp → Kitchen Lamp
Lamp → Hall Lamp

Lamp → LivingRoom Lamp
Radio → Bedroom Radio
Radio → Kitchen Radio
Radio → Hall Radio
Radio → LivingRoom Radio
Heater → Bedroom Heater
Heater → Kitchen Heater
Heater → Hall Heater
Heater → LivingRoom Heater

Change of Language in MIMUS

MIMUS controls the language switching by means of a special rule at dialogue level, where the language–dependent agents (ASR, TTS) as well as the lexicon and grammar modules are reconfigured.

```
( RuleID:    SWITCH;    /* Rule name. */
  TriggeringCondition:
    (DMOVE:specifyCommand,TYPE:SwitchLang);  /* DMove that triggers this rule. */
  DeclareExpectations: {
    Lang<=(DMOVE:specifyParameter,TYPE:Language);
                                          /* Expectation linked to the previous
                                          Dialogue Move.
                                          The rule won't apply the PostActions
                                          until the expectation is fulfilled.*/
    }
  ActionsExpectations: { /* Actions to be executed when an expectation is missing */
    [Lang] =>
        {ApplyTemplate(SwitchLanguage);} /* Which language do you want to speak? */
    }
  PostActions: {  /* Actions to be executed once all expectations are fulfilled */
     @if (@is-SWITCH.Lang.CONT == "english")
     {
        LoadNLU(English);       /* Load Lexicon and Grammar*/
        setGrammar(English);    /* ASR configuration */
        setLanguage(English);   /* TTS configuration */
     }
     @if (@is-SWITCH.Lang.CONT == "spanish")
     {
        LoadNLU(Spanish);       /* Load Lexicon and Grammar*/
        setGrammar(Spanish);    /* ASR configuration */
```

```
        setLanguage(Spanish);   /* TTS configuration */
     }
    ApplyTemplate(SwitchLangConfirm);    /* O.K., let's speak english/spanish now */
    }
)
```

4.5.5 Multimodality in MIMUS

As for multilinguality, multimodality will be considered from two points of view in MIMUS. First, we will show how multimodal grammars can be obtained through OWL2Gra, and then, how multimodal fusion is achieved. MIMUS allows fully multimodal interaction, ranging from speech–only to click-only productions, and any combination of these (multimodal fusion). Full multimodal ineraction also implies that MIMUS has the ability to generate accurate graphical and verbal answers (multimodal presentation). As previously explained, the USE information state or DTAC structure has been extended to allow for multimodal fusion. Regarding multimodal presentation, USE has defined a complete architecture at this level, based on three layers: a Content Planner, a Presentation Planner and a Realization Module.

Capturing Multimodality through OWL2Gra

Now let us assume the same scenario (i.e. the same ontology) but including multimodal entries; namely voice and pen inputs. Following Oviatt's results [20], it may be expected that the mixed input modalities (voice: *switch this on*, pen: click on the lamp icon) may also include alternative constituent orders, that is, different to the voice only input. The NLU module may therefore receive inputs such as: *lamp switch on* (verb at the end). [2]

This new set of rules can be easily accounted for by adding just one rule to the configuration file:

```
<rulesList>
   <forEach property="P"
            subPropertyOf="hasDeviceCommand">
      <forEach domain="X" range="Y">
         <rule>
            <left>Command</left>
            <right>P Y</right>
         </rule>
         <rule>
            <left>Command</left>
            <right>Y P</right>
         </rule>
      </forEach>
```

[2] Note that linguistically speaking this order is also possible in English in topicalized or left–dislocated constructions such as *The lamp, switch it on*.

```
   </forEach>
</rulesList>
```

The new output will be the same as before but including these new rules:

Command → Fan SwitchOff
Command → Heater SwitchOff
Command → Lamp SwitchOff
Command → DimmerLamp SwitchOff
Command → Radio SwitchOff
Command → TV SwitchOff
Command → Fan SwitchOn
Command → Heater SwitchOn
Command → Lamp SwitchOn
Command → DimmerLamp SwitchOn
Command → Radio SwitchOn
Command → TV SwitchOn
Command → Blind Close
Command → Blind Open

Multimodal Fusion Strategies

We have developed three different fusion strategies in TALK, and two of them have been implemented. The first solution is largely based on Johnston's work [11] [10], and involves modifying our parser to cope with simultaneous multimodal inputs, and to include temporal constraints at unification level. The second implementation proposes an original solution to the problem, and involves combining inputs coming from different multimodal channels at dialogue level. A third strategy has been recently presented [17], and to put it in few words, it combines the best features of the previous approaches.

A brief summary of these strategies is provided here, although a more detailed description is available in D1.2b [15].

Strategy 1

The first strategy implemented is based on Johnston's proposal, using a unification–based parser and including modality and temporal constraints at unification level. The MIMUS implementation differs from Johnston's in that a higher level of flexibility is provided.

The main motivation behind this strategy is that multimodality is conceived of as a single communicative act between two participants, and as such, it should be handled by a single multimodal grammar. This strategy is therefore implemented at NLU level (see figure 4.8). As expected,

MIMUS allows for the communicative act to range from speech–only to clicks–only or hybrid inputs, and all are considered equal as far as the grammar is concerned. This is an advantage as long as only single–task interactions are considered, leaving aside multiple task interactions. The pragmatic ambiguity which may result in multimodal multi–tasking cannot be resolved by a single grammar.

When the parser receives an input sentence (either speech–only, click–only or mixed-modality), it calls the lexical analyzer adding two new feature-value pairs: MODALITY, TIME_ST. These features are then used in conjunction with a set of logical operators to define complex expressions in order to enforce modality and temporal constraints.

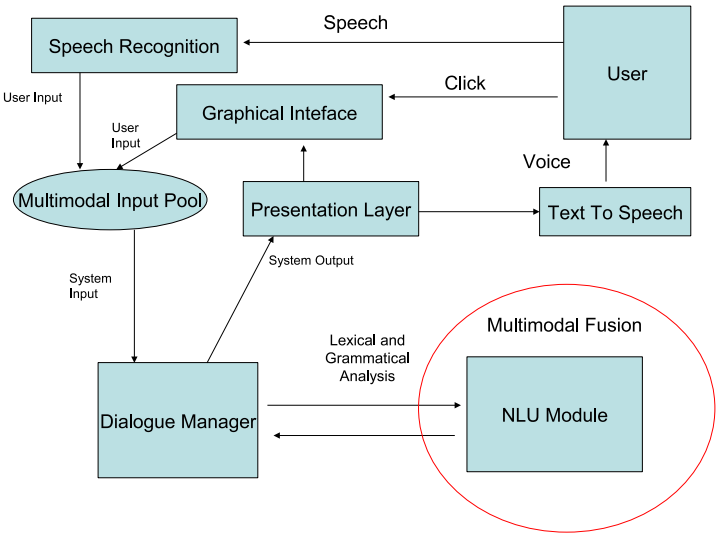

Figure 4.8: Strategy 1

Strategy 2

The second strategy combines simultaneous inputs coming from different channels (modalities) at Dialogue Level (see figure 4.9). The idea is to check the multimodal input pool before launching the actions expectations waiting an “inter–modality” time.
This strategy is implemented at dialogue level (Dialogue Manager Module) and assumes that each individual input can be considered as an independent Dialogue Move.
In this approach, the multimodal input pool receives and stores all inputs including information such as time and modality. The Dialogue Manager checks the input pool regularly to retrieve the corresponding input. If more than one input is received during a certain time frame, they are considered simultaneous or pseudo–simultaneous. In this case, further analysis is needed in order to determine whether those independent multimodal inputs are truly related or not.

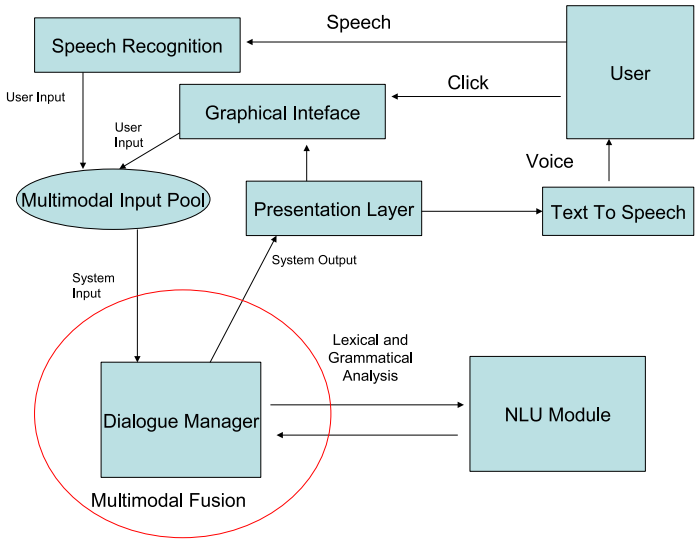

Figure 4.9: Strategy 2

Dialogue Rules may also be configured with the same logical operators mentioned in Strategy one, since the Dialogue Manager actually uses the unification module of the parser. Similar rules could be configured within the Dialogue Manager.

Basically, the difference lies in *where* these rules are applied: for Strategy one, the coverage is determined by the symbols (terminals and non terminals) within the grammar rules, while the coverage for Strategy two are the DTAC structures that describe the DMoves.

Strategy 3

Comparing the pros and cons of the previous strategies, it could be concluded that:

1. Strategy 1 is more coherent in terms of the definition of a communicative act as a single event that may be more or less complex (single vs. multiple modalities).

2. Nonetheless, strategy 1 implies a significant computational load and is more dependent on time measures, which is not the case in strategy 2. This dependency and precision need for strategy one implies as well larger amounts of real user data to tune the multimodal grammar.

3. When dealing with additional or alternative modalities, the inter-modality disambiguation will no longer be between pairs (one or the other), but would imply the generation of full disambiguation lattices. In this case strategy 2 would reach a significant degree of

complexity whereas strategy 1 could handle it more easily. Then again, there would be a significant computational overload with strategy 1.

4. Strategy 2 can handle independent simultaneous tasks in different modalities (multimodal multitasking), which would not be possible with strategy 1. Nonetheless, strategy 2 presents a potential theoretical problem that arises from the assumption that every uni-modal input can generate a dialogue move. No examples of this case have been found, but the opposite has not been proven either.

The goal of this new strategy is to take advantage of the positive side of each of the previous strategies: including multimodal grammar entries and temporal and modal constraints as in strategy 1, but delegating the decision to the dialogue manager, in order to take into account the additional information involved in the strategy 2 decision process. Fortunately, this can be done in MIMUS without much effort at all. The parser within MIMUS renders all possible parsing chunks. In previous versions of the system (speech-only versions), the most likely chunk would be selected by an internally developed criterion based on empirical data. However, when this selection strategy is deactivated, all possible parsing results are outputted. This basically means that given a grammar where simultaneous or pseudo-simultaneous multimodal entries may or not be related, the parser will output all possibilities: two unrelated events in different modalities, or one complex multimodal event. It will then be up to the dialogue manager to select which option is more likely to be appropriate, instead of having to build the most appropriate construction by a post-parsing unification process.

4.5.6 Multimodal Presentation in MIMUS

MIMUS offers graphical and voice output to the users through an elaborate architecture composed of a TTS Manager, a HomeSetup and GUI agents. The multimodal presentation architecture in MIMUS consists of three sequential modules. The current version is a simple implementation that may be extended to allow for more complex theoretical issues hereby proposed, and detailed in [27]. The main three modules are:

- Content Planner (CP): This module decides on the information to be provided to the user. It is encoded as attribute-value pairs in a variant of the DTAC protocol. As pointed out by [29], the CP cannot determine the content independently from the presentation planner (PP). In MIMUS, the CP generates a set of possibilities, from which the PP will select one, depending on their feasibility.

- Presentation Planner (PP): The PP receives the set of possible content representations and selects the “best” one in three steps:

    1. First, it checks the contents proposed against the available modalities, creating variants for those that are ambiguous, and discarding unfeasible options.
    2. Then, it uses manually predefined selection rules to restrict the set of possible presentations.

3. Finally, it checks whether there are concurrent options yet, in which case it applies an optimization algorithm based on [31] to select one of them.

Along these steps, the PP uses the following external knowledge resources, all of them encoded as OWL ontologies and accessed through the Knowledge Manager (KM): a Modality Model (based on Bernsen taxonomy, [4]), a User Model, a Context Model, a set of Multimodal Election rules and the Dialogue History.

- Realization Module (RM): This module simply takes the presentation generated and selected by the CP-PP, divides the final DTAC structure and sends each substructure to the appropriate agent for rendering.

4.6 Dialogue Examples

The following is a single dialogue currently supported by MIMUS. Before each utterance, a brief explanation of the phenomena illustrated is provided:

- HOTWORD: Shows how the system remains asleep until the key word is pronounced.
  - (Voice Input) Ambrosio
  - (Voice Output) A su servicio... (at your service)
- DEVICE MANAGER AND TALKING HEAD: Shows the system at work. Ambrosio nods while executing the physical command
  - (Voice Input) Enciende la cocina (switch on the kitchen)
  - (Visual Output) Expression–NOD
- MULTIMODAL SYMMETRY: Shows how the same task can be accomplished graphically
  - (Click Input) CommandOn - Patio_1
  - (Visual Output) Expression-NOD
- MULTICOMMAND: Shows how the user can say two commands in a single utterance
  - (Voice Input) Enciende la entrada y sube la persiana (switch on the hall and raise the blind)
  - (Visual Output) Expression–NOD (2)
- CONTEXT CHANGE: Shows how the OWL ontology is automatically changed to take into account a context change.
  - (Voice Input) ¿Cuántas luces hay encendidas? (how many lights are on?)
  - (Voice Output) Hay tres luces encendidas (there are three lights on)

- NATURAL LANGUAGE GENERATION: The system offers a more natural answer taking into account how the question is asked:
  - (Voice Input) ¿Cuántas luces hay encendidas en la cocina? (how many lights are on in the kitchen?)
  - (Voice Output) Hay una luz encendida en la cocina (there is one light on in the kitchen)
- COLLABORATIVE DIALOGUE: MIMUS asks for the piece of information missing.
  - (Voice Input) Apaga la luz (switch off the light)
  - (Voice Output) ¿Qué luz desea apagar? (which light do you wish to switch off?)
  - (Voice Input) La cocina (the kitchen)
  - (Visual Output) Expression–NOD
- MULTIMODAL FUSION: MIMUS fuses the information coming pseudosimultaneously if the inputs are complementary. The system does not ask for further information.
  - (Voice Input) Apaga esta luz (switch off this light)
  - (Click Input) Patio_1
  - (Visual Output) Expression–NOD
- MULTIMODAL MULTITASKING: The system does not fuse pseudo–simultaneous information if the inputs are NOT complementary.
  - (Voice Input) Enciende la cocina (switch on the kitchen)
  - (Click Input) CommandSwitch - TELEPHONE
  - (The Telephone GUI pops up)
  - (Visual Output) Expression–NOD
- RESTRICTIONS AND QUANTIFIERS: MIMUS handles quantifiers (all) and restrictions (except)
  - (Voice Input) Enciende todas las luces menos el dormitorio (switch on all the lights except the bedroom)
  - (Visual Output) Expression–NOD
  - (Voice Input) Apaga todas las luces menos el patio (switch off all the lights except the patio)
  - (Visual Output) Expression–NOD
- MULTILINGUALITY ON THE FLY: MIMUS can change languages at run–time:
  - (Voice Input) Cambia a inglés (switch to English)

  - (Voice Output) Very well. We'll speak English from now on.

- CONTEXT AWARENESS: MIMUS is context aware. The bathroom light is switched on without disambiguation.
  - (Click Input) CommandZoomIn - Bathroom_1
  - (Voice Input) Switch on the light
  - (Visual Output) Expression–NOD
  - (Click Input) CommandZoomOut

- DIALOGUE HISTORY AND RECOVERY ACTIONS: MIMUS recovers incomplete dialogue moves. The dialogue history is preserved through a language change.
  - (Voice Input) Switch on the light
  - (Voice Output) Which light do you want to switch on?
  - (Voice Input) Switch to Spanish
  - (Voice Output) Muy bien. Hablemos (Very well. We'll speak Spanish from now on) español a partir de ahora.
  - (Voice Output) Previamente indicó que quería encender una luz. 'Qué luz desea encender? (you previously requested to switch on a light. Which one do you mean?)
  - (Voice Input) El dormitorio (the bedroom)
  - (Visual Output) Expression–NOD

- MULTIMODAL FUSION OF MULTIPLE INPUTS: MIMUS fuses pseudo–simultaneous information including a multicommand.
  - (Voice Input) Apaga esta luz y enciende esta (switch off this light and switch on this one)
  - (Click Input) Patio_1
  - (Click Input) Kitchen_1
  - (Visual Output) Expression–NOD

- BYE!
  - (Voice Input) Adiós (goodbye)
  - (Voice Output) Si me necesita, llámeme. Ahora con su permiso, me retiro. (Alright then. Let me know if you need me. Bye for now!)

4.7 Conclusion

In this chapter, an overall description of MIMUS showcase for the in home domain has been provided, including the references as to where to find more detailed information. Throughout this project and in order to develop a fully multimodal and multilingual in–home showcase, a number of tasks have been carried out:

- Develop and implement multimodal and multilingual strategies
- Extend the IS to allow for multimodality
- Design 3 different fusion strategies
- Develop a fully functional home ontology in OWL
- Use the ontologies as the knowledge source for most tasks
- Develop a multimodal presentation architecture
- Implement a dynamic 3D Home Set-up
- Implement a 3D virtual character
- Design and implement a fully multimodal a multilingual WoZ experimental platform
- Conduct a series of multimodal experiments
- Collect MIMUS, a multimodal corpus for the in-home domain
- Use UCD (User Centered Design) strategies for the design and configuration of the MIMUS system
- Design and implement strategies to handle multimodal multitasking
- Design and implement the MIMUS showcase
- Other tasks such as the implementation of additional tools, wrappers, etc.

In relation to multimodality, two approaches to multimodal fusion have been implemented. Both approaches make use of the same extended information state reported in WP3. The first results have concluded that performing multimodal fusion at the dialogue level (by means of a multimodal input pool) poses some advantages over a grammar based multimodal fusion strategy. However, it is expected that the implementation of the third strategy should provide even more reliable results.

MIMUS approach to multilinguality is similar to the fusion strategy two, in the sense that both are controlled at dialogue level. There is a special dialogue rule in charge of controlling the language switching, where the ASR, the TTS and the NLU configurations are updated.

With regard to knowledge management, MIMUS performs reference resolution through RDQL queries over an OWL ontology describing the devices in the home. MIMUS has shown how the same ontology may be used in generating multilingual domain–specific grammars, thus allowing for a unified approach to multilinguality in the system. This approach is similar to that described for GoDiS in the GF paradigm, as described in [2].

The combined use of this unified approach to multimodality and multilinguality and the ISU approach has proved extremely profitable, and highly superior to current state–of–the–art dialogue systems: the naturalness and flexibility provided by the ISU approach combines with the rapid prototyping and coherence achieved through the use of a domain ontology in OWL. This approach will promise to be more fruitful if dialogue rules can be generated semi–automatically from the ontology as well.

Overall, MIMUS is a fully multimodal and multilingual showcase defined by a unified approach to multimodality and multilinguality in the Extended Information State Update approach. A number of theoretical and practical issues have been addressed successfully, resulting in a user–friendly, collaborative and humanized system.

Chapter 5

Conclusion

The work described in this deliverable has addressed the overarching research questions of how to unify multimodality and multilinguality in a common framework, what advantages this gives, how the Information State Update (ISU) approach can be used in this regard, and how to implement a unified approach. All these issues have been explored in relation to the in-home domain. The practical issue of implementation has been answered by our provision of three ISU-based showcase systems GoDiS, the Linguamatics Interaction Manager, and MIMUS, all illustrating various aspects of multimodality and multilinguality.

In this concluding chapter, we discuss our work on multimodality and multilinguality focusing on three issues: current state-of-the-art systems, the ISU approach, and the implementation of research in our showcase systems.

5.1 Advantages over current state-of-the-art

In a general way, the TALK systems showcased here add to the research and commercial applications fields by investigating the unification of multimodality and multilinguality in several interesting ways. The particular unification approach that we investigate is in itself an advantage over state-of-the-art systems, as demonstrated by the powerful, flexible, and coherent system behaviour that follows.

An specific advantage of GF for multimodal grammars is that input from different modalities can be combined, through the use of discontinuous constituents, in a way that is not possible in context-free grammars such as Regulus [25], except for through potentially costly post-processing.

Another advantage of our work is the combination of an ISU system such as GoDiS with a multilingual grammar framework such as GF, which enables rapid prototyping, and the porting of a dialogue system to a new language, a new domain, or a new modality.

A clear advantage of the systems showcased here over current industrial state-of-the-art systems, such as systems developed using VoiceXML, is the capabilities for advanced dialogue management, giving highly flexible and natural interactions with the user. This advantage pertains to

the ISU approach, as do a number of advantages over current state-of-the-art systems. Such advantages are described in the next section.

5.2 Advantages of the ISU approach

The ISU approach has proven highly advantageous in the development of a unified approach to multimodality and multilinguality, and for the systems showcased for the in-home domain here. A key to this is both the use of a central repository of information that is maintained throughout a dialogue, and the highly modular system architecture that comes with an ISU system.

The success of the ISU approach is quite generally shown by the successful porting of the systems showcased here, from unimodal and unilingual incarnations to today's fully flexible, multimodal and multilingual systems. This is shown by MIMUS through its inclusion of English and Spanish, a graphical display of an apartment, and a talking head. It is shown by GODIS through English and Swedish being included in all GODIS applications showcased, and a number of other languages incorporated in the GOTGODIS application, including the non-Indo-European language Finnish. The GODIS applications also include a number of different non-speech modalities. The success of the ISU approach is also shown by the Linguamatics Interaction Manager, where reconfigurability in relation to multimodality has been a major issue.

More specifically, the ISU approach enables the coding of dialogue behaviour independent of the languages and domains involved, so that dialogue behaviour can be developed separately from domain and linguistic knowledge, and dialogue behaviour components can be reused when the system is ported to another language or domain. This enables rapid porting of an ISU dialogue system to another domain as well as to a new language. It also allows multilingual systems, and the incorporation of several different domains within the same system, while maintaining coherence for system dialogue behaviour.

MIMUS and GODIS showcase the advantages of the ISU approach through the independence of dialogue behaviour from specific languages, both at the development stage, where dialogue behaviour can be modified and extended in isolation from linguistic resources and vice versa, and for the user at run-time. At run-time, MIMUS and GODIS both allow the user to switch languages in the middle of a task, and the information states and modular nature of both systems ensure that the task can be continued in the new language, all relevant information being maintained in the information state.

The porting of an ISU system to a new domain is illustrated in MIMUS through the inclusion of a number of different devices, functionalities, and types of dialogue, in the same system. In GODIS it is illustrated through the porting of the system to a number of different applications, and also through the possibility of application switching during a dialogue. The Linguamatics Interaction Manager here takes a similar approach to that used in MIMUS, through the interaction with a very large number of different devices enabled within the same system. In this regard, the Linguamatics Interaction Manager has also demonstrated real-life success of their system through its integration in an actual house.

For all three showcase systems, a central repository of information in the form of the information

state has provided an adaptive locus for multimodality, in that the same basic information state has been maintained for the move to multimodality, giving a "monotonic" implementation of a multimodal system from a unimodal system. For instance, in relation to MIMUS we have described how a pre-TALK unimodal DTAC structure has been augmented to a multimodal DTAC. In general this means that there has been no need to completely rewrite update rules and other dialogue system control, but that existing rules could be modified and new ones added, without the disruption of already existing system behaviour. Furthermore, for all three systems, a central repository of information in the form of a structured information state provides a coherent and accessible representation of the current context, as needed for the determination of multimodal and multilingual aspects.

The use of structured information states is also highly advantageous from an interactional perspective. An information state allows information to be accessed and modified through different means and in different orders, which provides a possibility for very advanced dialogue behaviour giving highly flexible dialogue systems. For instance, this is taken full advantage of in GODIS, which includes solutions for a number of dialogue challenges, such as grounding, feedback, clarification, multiple simultaneous tasks, information sharing between tasks, user initiative, belief revision, and so on. All these are enabled and given elegant solutions through the existence of a structured information state, and, importantly, the ISU approach also allows these dialogue processing factors to be solved domain-independently, so that they can be reused by any GODIS application. In fact, the GOTGODIS application is specifically designed with this in mind, as it ports the GOTTIS domain to GODIS, receiving all the GODIS dialogue solutions for free.

5.3 Implementation of research in the showcases

The present deliverable provides software for the implemented in-home showcases, with a focus on the implementation of multimodality and multilinguality. In this section we provide a concluding overview of how the research has been implemented in the three showcases.

The GODIS showcase system includes four different applications, and GODIS work is also supplemented by the separate system GOTTIS. The multimodal and multilingual GF grammars for the GODIS applications involve, first of all, a common GODIS resource grammar, that contains all contributions that are in common for GODIS applications. This common resource grammar is implemented using the GF Resource Grammar, which exists for several languages. Secondly, the GF implementation involves application-specific grammars. GOTTIS only contains such specific grammars, as it is not a GODIS application.

All GF grammars implement the unified approach to multimodality and multilinguality through the distinction between abstract and concrete syntax, where the concrete syntax corresponds to the different natural languages and the different modalities, all unified through the abstract syntax.

The implementation of the actual applications showcase the grammars and different input and output modalities, as well as dialogue management aspects. The GOTTIS system implements English and Swedish GF grammars. It implements integrated speech and pointing modalities

for user input, with the pointing being enabled through an electronic map. System output is implemented using the same map, as well as speech.

GOTTIS includes minimal dialogue management capabilities. The flexible and advanced capabilities enabled by the ISU approach are therefore implemented in all GODIS applications directly in GODIS, in the form of the information state, the update rules, and the overall system architecture and control. This includes extensions for multimodality and multilinguality. GODIS is in turn built using the TrindiKit4 for ISU modelling.

GOTGODIS, DJ-GODIS, and GODIS-DELUX all implement graphical input and output using the DynGUI, a generic GUI agent developed in TALK that enables the implementation of the research on the Multimodal Menu-based dialogue (MMD) approach. Graphical input and output in AGENDATALK is achieved through the stand-alone Borg calendar agent, which has been enhanced in TALK to allow advanced control of graphical output.

Speech input and output for GODIS is implemented using Nuance for ASR, and Vocalizer and Realspeak for TTS. OAA is used to wire together GODIS applications and the various components used for multimodality and multilinguality.

Table 5.1 shows an overview of the mapping between research issues and showcase GODIS applications. In addition to multimodality and multilinguality, the table includes information on the use of GF and features for dynamic reconfiguration (application switching and plug and play). The DICO application, developed outside of TALK but adapted for English in the project, is included for reference.

The Linguamatics Interaction Manager is designed to have a small footprint. The core code is implemented in C. Message passing uses a proprietary router written in Java. The Nuance Recogniser has been used for speech recognition, and Nuance Vocalizer for speech synthesis. For the installation at the Advantica home, Loughborough University supplied a graphical rendering program and a task manager which communicated with the devices. Communication with the Loughborough system is via message passing, with XML used for graphical output.

The MIMUS implementation focuses on light control over an interactive 3D floorplan of an apartment running on a Tablet–PC. Input may be by speech only, clicks only, or a combination of both. The user may interact in English, Spanish and German, although language change is only implemented for English into Spanish and viceversa. The three versions share the same OWL ontolgy, from which three different NLU grammars have been generated. Separate recognition grammars have also been developed for English, Spanish and German in Nuance format. A single set of dialogue rules is used for the three languages, as is the case of templates in the NL generation module. Output may be graphical and/or spoken, through a talking head and home setup specifically developed for the project. The TTS used for English and Spanish has been Loquendo, and Mary for German. The sofware has been implemented in C++, and agents communicate via OAA solvables.

Appendix A.1 lists all the relevant software for the three showcases.

	DJ-GoDiS	GoDiS-deLux	GoTGoDiS	AgendaTALK	DICO
Multimodal input	Yes, MMD	Yes, MMD	Yes, MMD	-	-
Multimodal output	Yes, MMD	Yes, MMD	Yes, MMD	Yes	Yes
Languages	Swedish, English	Swedish, English	Swedish, English, Finnish Italian French Spanish German	Swedish, English	Swedish, English
Language switching	Button click	Button click	Button click	Dialogue	Offline
Use of GF	Parsing, generation, ASR	Parsing, generation, ASR	Parsing, generation, ASR	ASR SLM	-
Application switching	Implicit	Implicit	(Implicit)	-	Explicit
Plug and play	Offline	Offline	(Offline)	-	-

Table 5.1: Mapping research to showcases for GoDiS applications.

Bibliography

[1] Tilman Becker, Nate Blaylock, Ciprian Gerstenberger, Andreas Korthauer, Nadine Perera, Peter Poller, Jan Schehl, Frank Steffens, Rosmary Stegmann, and Jochen Steigner. In-car showcase based on talk libraries. Deliverable D5.3, TALK Project, 2006.

[2] Tilman Becker, Staffan Larsson Peter Poller, Oliver Lemon, Guillermo Pérez, and Jan Scheh. D5.1: Software infrastructure. Deliverable 5.1, Talk Project, 2006.

[3] Tilman Becker, Peter Poller, Staffan Larsson, Oliver Lemon, Guillermo Pérez, Jan Schehl, Karl Weilhammer, and MORE AUTHORS. Software infrastructure. Deliverable D5.1, TALK Project, 2006.

[4] N. O. Bernsen. Multimodality in language and speech systems. from theory to design support tool. *Multimodality in Language and Speech Systems*, 2001.

[5] Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. Development of multimodal and multilingual grammars: viability and motivation. Deliverable D1.2a, TALK Project, 2005.

[6] K. Clarke, M.R. Lewin, D. Atkins, and R.S. Kalawsky. Testing a framework for multimodal control in the home environment. In *Proc. Perspectives in Pervasive Computing*, pages 87–95, IEE, London, 2005. DTI.

[7] Ivana Kruijff-Korbayová (editor), Gabriel Amores, Nate Blaylock, Stina Ericsson, Guillermo Pérez, Kalliroi Georgila, Michael Kaisser, Staffan Larsson, Oliver Lemon, Pilar Manchón, and Jan Schehl. Extended information state modeling. Deliverable D3.1, TALK Project, 2005.

[8] Ivana Kruijff-Korbayová (editor), Gabriel Amores, Johan Bockgård, Stina Ericsson, Ciprian Gerstenberger, Rebecca Jonson, Oliver Lemon, Pilar Manchón, David Milward, Peter Poller, Aarne Ranta, and Jan Schehl. Modality–specific resources for presentation. Deliverable D3.3, TALK Project, 2006.

[9] Stina Ericsson, Ciprian Gerstenberger, Pilar Manchón, and Jan Schehl (editor). Plan library for multimodal turn planning. Deliverable D3.2, TALK Project, 2006.

[10] Michael Johnston. Unification–based multimodal parsing. In *Coling–ACL*, pages 624–630, 1998.

[11] Michael Johnston, Philip R. Cohen, Sharon L. Oviatt David McGee, James A. Pitman, and Ira A. Smith. Unification–based multimodal integration. In *ACL*, pages 281–288, 1997.

[12] Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, Göteborg University, 2002.

[13] Staffan Larsson, Gabriel Amores, Rebecca Jonson, and Jose Qesada. Siridus system architecture and interface report (enhanced version). Project deliverable 6.3, SIRIDUS, 2002.

[14] Peter Ljunglöf, Gabriel Amores, Håkan Burden, Pilar Manchón, Guillermo Pérez, and Aarne Ranta. Enhanced multimodal grammar library. Deliverable D1.5, TALK Project, 2006.

[15] Peter Ljunglöf, Gabriel Amores, Robin Cooper, David Hjelm, Pilar Manchón, Guillermo Pérez, and Aarne Ranta. Multimodal grammar library. Deliverable D1.2b, TALK Project, 2006.

[16] Peter Ljunglöf, Björn Bringert, Robin Cooper, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonsson, Staffan Larsson, and Aarne Ranta. The TALK grammar library: an integration of GF with TrindiKit. Deliverable D1.1, TALK Project, 2005.

[17] P. Manchon, G. Perez, and G. Amores. Multimodal fusion: A new hybrid strategy for dialogue systems. In *Proceedings of International Congress of Multimodal Interfaces (ICMI06)*, Banff, Alberta, 2006.

[18] David Milward, Gabriel Amores, Tilman Becker, Nate Blaylock, Malte Gabsdil, Staffan Larsson, Oliver Lemon, Pilar Manchón, Guillermo Pérez, and Jan Schehl. Integration of ontological knowledge with the isu approach. Deliverable D2.1, TALK Project, 2005.

[19] David Milward, Gabriel Amores, Nate Blaylock, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, and Guillermo Pérez. Dynamic multimodal interface reconfiguration. Deliverable D2.2, TALK Project, 2006.

[20] Sharon Oviatt, S. L., DeAngeli, A., and Kuhn K. Integration and synchronization of input modes during multimodal human–computer interaction. In *Proceedings of Conference on Human Factors in Computing Systems: CHI '97.*, 1997.

[21] J. F. Quesada and J. G. Amores. Knowledge–based reference resolution for dialogue management in a home domain environment. In M. Ellen J. Bos and C. Matheson, editors, *Proceedings of the sixth workshop on the semantics and pragmatics of dialogue (Edilog)*, pages 149–154, September 2002.

[22] José F. Quesada, Doroteo Torre, and J. Gabriel Amores. Design of a natural command language dialogue system. Project deliverable 3.2, SIRIDUS, 2000.

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

[24] M. Rayner, D. Carter, P. Bouillon, V. Digalakis, and M. Wirén. *The Spoken Language Translator*. Cambridge University Press, 2000.

[25] Manny Rayner, Beth Ann Hockey, and Pierrette Bouillon. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI Publications, 2006.

[26] T. Russ, A. Valente, R. MacGregor, and W. Swartout. Practical experiences in trading off ontology usability and reusability. In *Proceedings of the Knowledge Acquisition Workshop (KAW99)*, Banff, Alberta, 1999.

[27] Jan Schehl, Gabriel Amores, Stina Ericsson, Ciprian Gerstenberger, and Pilar Manchón. Plan library for multimodal turn planning. Deliverable 3.2, Talk Project, 2006.

[28] W3C. Simple knowledge organisation system (skos), 2006. http://www.w3.org/2004/02/skos/core/.

[29] W. Wahlster, E. Andre, W. Finkler, H. Profitlich, and T Rist. Plan-based integration of natural language and graphics generation. *Artificial intelligence*, pages 287–247, 1993.

[30] Karl Weilhammer, Rebecca Jonson, Aarne Ranta, and Steve Young. Generation of language models using GF. Deliverable D1.3, TALK Project, 2006.

[31] M. X. Zhou and V Aggarwal. An optimization-based approach to dynamic data content selection in intelligen multimedia interfaces. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, 2004.

Appendix A

Software Library

This chapter lists the software in the deliverable. With the systems in alphabetical order, Section A.1 gives the software for GODIS, Section A.2 the software for the Linguamatics Interaction Manager, and Section A.3 for MIMUS.

A.1 Software for GODIS

This section lists the software for the GODIS applications GODIS-DELUX, DJ-GODIS, GOT-GODIS, and AGENDATALK, and for the GOTTIS system. Following the structure of Chapter 2, grammars are listed first, in A.1.1. Software for the applications are then listed in A.1.2.

A.1.1 Grammars

Grammars for GOTTIS

In addition to the grammars listed below, there are a number of modules which link them together.

- **Common grammars**
  - `Transport`: Defines the category of transit network stops.
  - `Lines`: Defines the category of transit network lines.
- **City-specific grammars**
  - `Gbg`: Defines the Gothenburg transit network stops.
  - `GbgLines`: Defines the Gothenburg transit network lines.
- **System grammars**
  - `Route`: Abstract module which defines the routes given as answers by the system.

  - `RouteMap`: Concrete syntax for drawing output.
  - `RouteEng`, `RouteSwe`: Concrete syntax for speech output.

- **User grammars**

  - `TransportQuery`: Defines the (multimodal) queries which the user can give to the system.

Common GODIS grammars

The grammars are divided into grammars for System utterances, for User utterances, and general resources.
A typical GF grammar (say *Gram*) consists of one abstract syntax which is called *Gram* and a number of concrete syntaxes. One of the concrete syntaxes contains the GODIS semantics, this will be called *Gram*Sem. Then there is a number of syntaxes for the different languages of the domain, e.g., *Gram*Eng, *Gram*Swe, *Gram*Spa, ... If the grammar is written as a language-independent grammar, it is called *Gram*I. Finally the multimodal system grammars are called *Gram*MMI, *Gram*MMEng, *Gram*MMSwe, etc. In the following we list all these grammars as one single multilingual grammar, which we call *Gram*. Note that one such cluster does not necessarily contain all the mentioned files, it depends on the grammar.

- **General resources**

  - `Prolog`: Contains resources for building terms in Prolog syntax
  - `GodisApp`: Contains resources for combining several applications
  - `GodisMM`: Contains resources for multimodality, both input and output
  - `GodisResource`: Contains general language-independent resources
  - `GodisLexicon`: Contains general language-specific resources
  - `GodisLang`: More language-independent and language-specific resources

- **System grammars**

  - `GodisCat`: Contains (language-independent) definitions of the categories used in system utterances
  - `GodisSystem`: Contains (language-specific) definitions of domain-independent system utterances

- **User grammars**

  - `GodisUser`: Contains definitions of domain-independent categories and user utterances

Grammars for GoTGoDiS

- **General resources**
  - `Lines`: Contains definitions of trams, busses, and ferry lines
  - `Stops`: Contains definitions of tram/bus/ferry stops
  - `TramLexicon`: Contains language-specific definitions of lexical items
- **System grammars**
  - `TramSystem`: Contains language-independent definitions of domain-specific system utterances
- **User grammars**
  - `TramUser`: Contains language-independent definitions of domain-specific user utterances

Grammars for AgendaTalk

- **General resources**
  - `BookingDates`: Contains definitions of the concept of Date.
  - `BookingTimes`: Contains definitions of the concept of Time.
  - `BookingEvents`: Contains definitions of the concept of Event.
  - `Booking`: The union of BookingDates, BookingTimes and BookingEvents.
  - `AgendaLexicon`: Contains language-specific definitions of lexical items.
- **System grammar**
  - `AgendaSystem`: Contains definitions of domain-specific system utterances
- **User grammar**
  - `AgendaUser`: Contains definitions of domain-specific user utterances

Grammars for DJ-GoDiS

- **General resources**
  - `MusicArtists`: Contains definitions of artists
  - `MusicSongs`: Contains definitions of songs
  - `Music`: The union of all artists and songs
  - `MP3Lexicon`: Contains language-specific definitions of lexical items

- **System grammars**
  - `MP3System`: Contains language-independent definitions of domain-specific system utterances
  - `MP3SystemDelux`: The union of the System grammars of this application and GODIS-DELUX
- **User grammars**
  - `MP3Global`: Contains definitions of domain-specific user utterances, which are global and accessible by other applications
  - `MP3User`: Extends the global utterances with utterances which are local to this application
  - `MP3Delux`: The union of this application's local grammar (MP3User), and the GODIS-DELUX global grammar (DeluxGlobal)

Grammars for GODIS-DELUX

- **General resources**
  - `Lamps`: Contains definitions of different kinds of lamps
  - `Rooms`: Contains definitions of the available rooms
  - `Socket`: Contains definitions of the available lamps in each room
  - `DeluxLexicon`: Contains language-specific definitions of lexical items
- **System grammars**
  - `DeluxSystem`: Contains language-independent definitions of domain-specific system utterances
  - `DeluxSystemMP3`: The union of the System grammars of this application and DJ-GODIS
- **User grammars**
  - `MP3Global`: Contains definitions of domain-specific user utterances, which are global and accessible by other applications
  - `MP3User`: Extends the global utterances with utterances which are local to this application
  - `MP3Delux`: The union of this application's local grammar (DeluxUser), and the DJ-GODIS global grammar (MP3Global)

A.1.2 Applications

GOTTIS

The GOTTIS software can be found as a `tramdemo` library, separate from the GODIS applications.

GOTGODIS

GOTGODIS software can be found in the `domain-tram` library. This library contains the domain modules `tram-multimodal-dme.pl` and `tram-multimodal-control.pl` and configuration files needed to start the application. The GF grammars used for parsing and generation and the speech recognition grammars are found in the `grammars` directory.

The domain dependent resources used in GOTGODIS are located in the directory `Resources`. Here we find the device files `device_graph.pl` and `device_map.pl`, the file `domain_tram.pl` and `semsort_tram.pl`.

To run the application you would also need the file `tramdemo.jar` which includes the OAA map agent and OAA graph agent.

AGENDATALK

All AGENDATALK software can be found in the `domain-agendatalk` library. This includes domain knowledge and lexica in the library `Resources`, as well as device files for the Borg calendar. It also included speech files, and modules for the advanced generation capabilities in AGENDATALK.

DJ-GODIS

The DJ-GODISsoftware is in the `domain-player` library. This library contains domain modules, grammars, various resources, and all other software for the application.

GODIS-DELUX

GODIS-DELUX sofware can be found in the `domain-delux` library. This library contains the domain modules `delux-multimodal-dme.pl` and `delux-multimodal-control.pl` and configuration files needed to start the application. The GF grammars used for parsing and generation and the speech recognition grammars are found in the `grammars` directory.

The domain dependent resources used in GODIS-DELUX are located in the directory `Resources`. This is the location of the device files for each lamp and for the database device. Located here is also the prolog database containing specification of the GODIS-DELUX home (`database.pl`). We also find the files `domain_delux.pl` and `semsort_delux.pl`.

Files needed for the GODIS-DELUX gui is found in the directory `Delux_gui`. This includes the OAA GODIS-DELUX agent used in the application.

A.2 Software for the Linguamatics Interaction Manager

The Linguamatics Interaction Manager consists of the following modules:

1. A Java router
2. A 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) The ontology

A.3 Software for MIMUS

This section lists the software and resources included as a part of MIMUS, each subsection corresponding to a different directory. The structure and names refer to the CD attached to this deliverable. The software components are OAA agents described in deliverable 5.1 [2].

A.3.1 Root Directory: Batch Files

At the root directory there is a set of initial batch files (*.bat). Their names are variations over "domo–Talk" whose extension depends on the language in which we want to use MIMUS ("en" = English, "es" = spanish, "ger" = german) and whether we want full trace active or not ("test").

The Base_Directory path has to be modified to specify where the MIMUS directory structure is placed and where the OAA facilitator is installed.

A.3.2 VRM

VRM stands for "Voice Recognition Manager". This directory includes the Nuance Wrapper. Nuance 8.5 must be installed in the computer where the software is executed, and the License Manager script has to be modified, filling the field "PUT-YOUR-LICENSE-HERE".

A.3.3 Talking Head

MIMUS avatar, expressing emotions graphically and synthesizing voice through Loquendo TTS. Needs VS.Net framework and Loquendo 7.0.1 installed.

A.3.4 Ontology

OWL ontology with all the in-home knowledge configured as triplets Subject–Property–Object.

A.3.5 MMInputPool

External pool for multimodal inputs storage. Provides the elements upon request acting as a FIFO queue.

A.3.6 MimusCore

This directory includes the MIMUS Dialogue Manager. A subfolder named “ConfigFiles” includes the lexicon, grammar and dialogue configuration files, classified by language. There is a special folder called “common_rules” with dialogue rules that apply to all languages.

A.3.7 Merlin

As an alternative to the Talking Head for those users without Loquendo installed, the agent “Merlin” is included, a wrapper for the Microsoft Animated Agent that uses the TTS provided with Windows XP. Needs the Microsoft Animated Agent software installed (free with Windows XP).

A.3.8 HomeSetup

This agent represents the house layout, loaded from the Ontology at runtime. Needs VS.Net framework installed.

A.3.9 jDeviceManagerAgent

Agent which translates the software commands to physical ones through the X10 protocol. Needs the HomeControl software installed.

A.3.10 jKManagerAgent

Agent that queries the OWL ontology by means of the RDQL Language.

A.3.11 jDisplayAgent

Agent whose role is to present graphically the system outputs, either by text or by a list of clickable options. The decision on how to present the information is taken by the MIMUS Dialogue Manager.

A.3.12 jMenuAgent, jMP3Agent, jTelephoneAgent

The idea behind these agents is to provide a centralized control of more complex devices that are likely to be found in the house, like a telephone or an MP3 player. However, in its current status, only the graphical representation of these modules is shown, with no further funcionality.