D5.1: Software Infrastructure

Tilman Becker (ed), Peter Poller, Staffan Larsson,
Oliver Lemon, Guillermo Pérez, Karl Weilhammer

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable 5.1

January 15, 2007

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

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

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

Security	Public
Contractual date of delivery	M36 = December 2006
Actual date of delivery	January 15, 2007
Deliverable number	5.1
Deliverable title	D5.1: Software Infrastructure
Type	Prototype and Documentation
Status & version	Final January 15, 2007
Number of pages	62 (excluding front matter)
Contributing WP	5
WP/Task responsible	DFKI
Other contributors	Stina Ericsson, David Hjelm, Rebecca Jonson, Jan Schehl Ivana Kruijff-Korbayová, Nate Blayock, Daniel Bobbert
Author(s)	Tilman Becker (ed), Peter Poller, Staffan Larsson, Oliver Lemon, Guillermo Pérez, Karl Weilhammer
EC Project Officer	Evangelia Markidou
Keywords	Software Infrastructure

The partners in TALK are:

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

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

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

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

Contents

Summary

This report details all the software infrastructure work done in workpackage 5 “Infrastructure and Integration” for task 5.1 “Infrastructure” of the TALK project. This concerns two important aspects: i) the provision of an appropriate common software infrastructure framework for efficient inter-modular data communication (i.e. middleware) to be used for the various multimodal dialogue systems implemented in the project, and ii) the provision of the respective individual middleware-compliant software components which were used for the individual dialogue systems.

Chapter 1 gives an introduction to the single subtasks of Task 5.1 of the project.

In chapter 2, we recapitulate the major steps conducted during the project concerning the provision of an appropriate, commonly used intermodular communication infrastructure and architecture (middleware). Starting with the identification of relevant software infrastructure candidates, we carefully identified the optimal candidate, the Open Agent Architecture (OAA). Finally, we extended and improved it with respect to project-specific needs as they were requested by partners during the project.

The focus of this deliverable lies on the work that has been done during the project. However, all TALK showcase systems fall back on software components whose functionality or even their OAA-compliance already existed before the project. For each of the components it is clearly stated how the work done within the TALK project relates to what already existed before the project. Furthermore, in order to separate individually OAA-wrapped TALK showcase system components from more generic multiply used components or software frameworks we have split them up into different chapters.

In chapter 3, we describe the work on multiply used/usable generic or general purpose modules (with OAA wrappers) or software frameworks that were developed within the TALK project and (in contrast to below) used for/in multiple systems.

Chapter 4 provides a compilation of the individual software components (modules) used in TALK showcase systems whose basic functionality either already existed before the project or was newly implemented during the project.

Chapter 5 briefly describes some of the dialogue systems that were built within the TALK architecture and shows for each case which of the components described in the previous two chapters were used and how. Detailed information on the systems themselves can be found in the deliverables of the respective workpackages in which they were developed and implemented.

In Chapter 6, we conclude the software infrastructure work of the project and outline future directions for further simplifying software infrastructure issues in distributed implementations and environments.

Chapter 1

Introduction

This deliverable provides a collection of all the work done within Task 5.1 over the three years of the project.

The main objective of WP5 as a service package was providing a common software basis for libraries that are developed in WPs 1-4 and their integration into laboratory systems. Task 5.1 was to provide a software infrastructure of supporting modules with common interfaces. To this end, the TALK project has chosen a flexible architecture to integrate new and existing modules into different showcase systems and provides generic modules that were used in multiple systems.

We have thus identified the following three major tasks to provide appropriate software infrastructure:

1. Identification of a suitable middleware connecting the components of a system. In the first year, we have settled on using the Open Agent Architecture (OAA). Chapter 2 describes the decision process that has let to the selection of OAA. The list of systems described in chapter 5 demonstrates the success of this choice.

2. Provide generic, general purpose modules (i.e., generic, application independent compontents, tools or frameworks). These modules are described in chapter 3.

3. Make existing (and new) modules accessible for TALK showcase systems based on this middleware (as OAA wrappers). Chapter 4 collects these modules.

The approach taken in the TALK project to modularization and reuse of modules through a common middleware has substantially helped the development of showcase systems. In these systems, many modules are needed that are not derived by the core research and development efforts of partners, e.g., speech recognition and speech synthesis modules, and of course the backend applications such as calendars, data bases and MP3 players. The choice of a common middleware, i.e., the OAA architecture has allowed us to quickly wrap existing software for integration in OAA and then reuse these wrappers in multiple systems. This proved particulary useful for speech related and database modules.

The following chapters provide detailed information on (i) the OAA architecture as it was used in TALK, (ii) the generic general purpose components and software frameworks developed and multiply used within the TALK project, (iii) individual software components used in TALK showcase system, and (iv) some of the dialog systems build out of the aforementioned software infrastructure. Finally, we conclude the overall outcome of the sofware infrastructure task of the TALK project.

Chapter 2

Middleware: The Open Agent Architecture OAA

The common basis of all systems developed within the TALK project is that they use the OAA architecture as the intermodular communication infrastructure. As described in detail in D5.1.1 the selection of OAA as a common software infrastructure used throughout the project was well funded by (i) the extensive experience of project partners with OAA, and (ii) the already existing and re-usable software infrastructure work of them.

In this chapter, we briefly report on the latest news about OAA from SRI as well as individual improvements/extensions that were implemented within TALK due to corresponding requirements within the context of one of the TALK systems.

2.1 Enhancements by SRI

SRI has released its new version 2.3.1 of the OAA package on November 5, 2005. The changes from version 2.3.0 mostly comprise bug fixes within libraries, extended and simplified (API) documentation, as well as simplifications and improvements concerning handling and control of OAA-based systems. TALK showcase systems were not affected by this update.

2.2 Enhancements from the TALK project

In the course of development and implementation of the DFKI/USAAR in-car baseline and final showcase systems, DFKI and USAAR implemented several extensions concerning the automatic data logging by the OAA monitor agent as well as better supporting the start-up and control of distributed agent communities by the StartIt agent. The logging facilities are vital for documentation and evaluation of the data collected in the evaluation experiments performed with the in-car baseline system and the in-car final showcase system.

2.2.1 Extensions of the OAA Monitor

1. In the course of preparing the DFKI/USAAR SAMMIE WOZ experiment system we extended the log messages of the OAA monitor. Each message was supplemented by a timestamp in milliseconds. As this is done within the OAA monitor, the timestamps represent the exact point in time at which the respective message was recognized by the OAA monitor. The precise timestamps were initially needed for the annotation of the WOZ system interactions into the NXT format. After the final delivery of the SAMMIE baseline in-car system, we developed and implemented an automatic profiling tool based on these timestamped OAA monitor messages. The profiling tool was used to (i) separate the runtime needed for message passing via OAA from the runtime of the individual modules themselves, and (ii) to separate the time consuming modules from the efficient ones. For the baseline system, it turned out that the dialogue manager was the by far most time consuming component which led us to a different system architecture consisting of a series of more efficient PATE-based components communicating outside OAA (see D5.3 for details).

2. For the DFKI/USAAR in-car baseline system we implemented a flexible automatic logging facility as an addition to the already existing monitoring facilities of the OAA monitor. The logging procedure is realized as follows: As soon as the “start new dialog” button is clicked the OAA monitor generates a new logging directory containing a timestamp in its name. The logging itself runs in parallel to the monitoring steps. First, two new filenames for the different logging levels are generated. Then, each time a new OAA message is monitored, this message is also automatically saved within the corresponding logfile. Finally, we extended the OAA monitor window such that this automatic OAA message file saving procedure is done by default but may optionally be turned off within the Options menu. For the SAMMIE final showcase system evaluation, this tool was replaced. In order to leave the decelerating OAA monitor out of the system components, we implemented a special, multi-functional software tool for the final showcase evaluation which (i) very specifially logs only evaluation relevant data of system interactions, (ii) generates easy to read html pages that contain descriptions of the logged system interaction events, and (iii) provides the experiment conductor an easy to use tool for automatic integration of previously defined events or notes into the logdata based on respectively associated function keys.

3. As the automatic logfile saving binds additional resources of the OAA monitor we wanted to limit other logging functionalities if possible. We found that the local buffers of the OAA monitor are filled without any limit. Consequently, we implemented an automatic buffer limitation routine for the OAA monitor which delimits the locally stored number of messages by 1000. Now, each time the 1001st message is added, the oldest message is automatically deleted within the same step. As with the automatic logfile saving, this automatic buffer limitation routine is selected by default but may also optionally be turned off in the Options menu.

2.2.2 Supporting Distributed Agent Communities by the StartIt Agent

As the in-car baseline system turned out to be very complex by the number of agents alone, we improved the StartIt agent facilities to run (some of the) agents on remote machines. Although remote processing needs several preparations before (ssh without passord, X11-server, and cygwin if running on a windows machine) this significantly simplifies handling and managing distributed system configurations.

Chapter 3

General Purpose Modules and Frameworks

This chapter describes components, ranging from singular agents to complete frameworks, that can and have been used in multiple TALK systems. Some of these components have been specifically developed for TALK while others have been substantially enhanced over previous versions.

3.1 Dipper

The DIPPER (Dialogue Prototyping Equipment and Resources) architecture is a collection of software agents for prototyping (spoken) dialogue systems implemented on top of the Open Agent Architecture (OAA). DIPPER is not a dialogue system itself, but DIPPER supports building (spoken) dialogue systems, by offering interfaces to speech recognisers (ATK, Nuance), speech synthesisers (Festival), parsers (GEMINI, REGULUS) and other kinds of agents. See also [1] and [2].

Although DIPPER supports many off-the-shelf components useful for spoken dialogue systems (such as the aforementioned ones), it comes with its own dialogue management component (DIPPER DME), based on the information-state-update approach to dialogue modelling.

The DIPPER DME component borrows many of the core ideas of the TrindiKit, but is stripped down to the essentials, uses a revised update language (independent of Prolog), and is more tightly integrated with OAA. The DIPPER DME is written in Sicstus Prolog. Moreover, a new version has been implemented in Java. Both versions come with OAA wrappers.

A complete dialogue system can be implemented using DIPPER DME, OAA and a collection of agents, which includes: (1) agents for input/output modalities, (2) agents for the dialogue move engine, and (3) supporting agents. The DIPPER DME is the core of the system controlling the flow of information among the agents. The OAA term “solvable” is used to describe the services offered by agents. Each agent may have more than one solvable. The DIPPER DME has to call a solvable in order to give input or get output from an agent.

To run the DIPPER DME the user must define the following files: `INFO-STATE.is` is a Prolog file containing the definition of the information state, `UPDATE-RULES.urules` is a file with the update rules for that information state, `RESOURCES.pl` (optional) is a file with further Prolog definitions (specific for the application).

The DIPPER environment provides a Graphical User Interface (GUI) that assists during development (Figure 3.1). This GUI starts and stops the DIPPER DME and keeps a history of updates. In addition, the developer is able to engage in "time-travelling", by backtracking in the dialogue and restarting the dialogue from any point in the past. The 'Step' function applies just one update rule before returning control to the GUI and is particularly helpful in verifying the intended effect of an update rule. The 'Spy' function displays all rules that are satisfied by the current information state. Both functions can be used for step by step monitoring the triggering and execution of update rules.

Figure 3.1: The DIPPER Graphical User Interface.

The solvables of an agent are implemented by function calls (in C++ and Java) or predicate definitions (in Prolog) by the agents that provide them. DIPPER has beem maintained via the TALK project, and used in the D4.2 TownInfo system, [11, 10] and a small modification was made to allow running without GUI updates, for increased speed.

Use and Licenses

It is used in many projects for developing dialogue applications at the University of Edinburgh. DIPPER is freely available for research purposes and it has been downloaded by other research groups (there are 264 downloads by 6/12/2006).

Links

- DIPPER versions can be downloaded from `https://www.inf.ed.ac.uk/research/isdd/`

3.2 GoDiS

GoDiS is a dialogue system implemented using TrindiKit. The central components are the GoDiS DME (Dialogue Move Engine) and the GoDiS IS (Information State which implement Issue-based dialogue management [8].

The runtime behavior of the GoDiS DME and IS depends in part on the application, but roughly, the DME updates the IS based on observed dialogue moves and selects appropriate moves to be performed by the system based on the IS. The dialogue management capabilities of GoDiS include dealing with multiple simultaneous tasks, information sharing between tasks, question reraising, question accommodation and reaccommodation, simple plan recognition, three-level grounding and feedback management, sequencing management, and menu-based dialogue.

The user definable knowledge sources for a typical GoDiS application is a GF application grammar, domain knowledge (including ontology and dialogue plans) and a device interface to connect some outside device to the system.

Use and Licenses GoDiS is freely available for download for non-commercial applications. It has been used at Göteborg University as well as internationally in research and education.

Links

- GoDiS can be downloaded from `http://www.ling.gu.se/grupper/dialoglab/godis/`

3.3 TrindiKit 4

TrindiKit is a toolkit for building and experimenting with dialogue move engines and information states, that has been developed in the TRINDI, SIRIDUS and TALK projects. We use the term information state to mean, roughly, the information stored internally by an agent, in this case a dialogue system. A dialogue move engine, or DME, updates the information state on the basis of observed dialogue moves and selects appropriate moves to be performed.

The information state update approach to dialogue management views utterances as dialogue moves which update, and are selected on the basis of, a structured information state by means of update rules. As such, this approach is fairly general and allows the implementation of many different theories of dialogue.

Apart from proposing a general system architecture, TrindiKit also specifies formats for defining information states, update rules, dialogue moves, and associated algorithms. It further provides a set of tools for experimenting with different formalizations of implementations of information states, rules, and algorithms. To build a dialogue move engine, one needs to provide definitions of update rules, moves and algorithms, as well as the internal structure of the information state. One may also add inference engines, planners, plan recognizers, dialogue grammars, dialogue game automata etc..

In TrindiKit4, the system components, e.g. Total Information State, modules, resources, controller, can be distributed across several OAA agents. Each agent publishes its capabilities as OAA solvables, so that other OAA agents can call make use of them. This allows for using only a part of the TrindiKit functionality in a system, since overall system control can be put outside TrindiKit itself.

The dialogue system designer can define or use existing TrindiKit agents which communicate via OAA. The TrindiKit controller can run several control modules in parallel, coordinating the work of modules. The TrindiKit agents work together as one TrindiKit system.

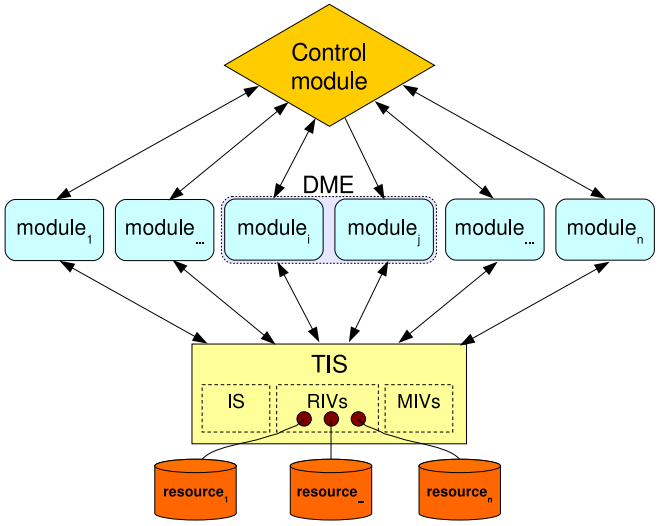


Figure 3.2: Schematic overview of TrindiKit

The knowledge sources for a typical TrindiKit4 system are system properties, IS type specification, resources and resource interfaces, module interfaces, dialogue moves, update rules, update and control algorithms, and additional modules and resources. Together, these knowledge sources make up a more or less complete dialogue system.

In TrindiKit4, the system components, e.g. Total Information State, modules, resources, controller, can be distributed across several OAA agents. Each agent publishes its capabilities as OAA solvables, so that other OAA agents can call make use of them. This allows for using only a part of the TrindiKit functionality in a system, since overall system control can be put outside TrindiKit itself.

There are four main component types in TrindiKit:

- Controller: executes control algorithms
- Module: executes module algorithms
- TIS: executes TIS checks and updates
- Resources: can be hooked up to the TIS via resource interface variables, defines resource relations and operations

Each component type, except for resources, has its own interface which is used to make use of the components capabilities.

Solvables All TrindiKit components can be called from OAA provided that the TrindiKit agent has connected to OAA (i.e. property `oaa` is set to `yes`). The TrindiKit solvables are as follows:
Control agents (TrindiKit agents with property `control` set to `yes`) declare:

- `tkit_trigger(TC)`, which corresponds to `tkit_control_access` predicate `trigger/1`
- `tkit_control`, which corresponds to `tkit_control_access` predicate `control/0` (not yet implemented)

For each of the agent's module algorithms a solvable of the form

- `tkit_call_module(M,A)`

where `M` is the name of the module and `A` is the algorithm, is declared.
TIS agents (TrindiKit agents with property `tis` set to `yes`) declare:

- `tkit_check(C)`, which corresponds to `tkit_tis_access` predicates `check_condition/1` and `check_conditions/1`
- `tkit_apply(O)`, which corresponds to `tkit_tis_access` predicates `apply_update/1` and `apply_updates/1`
- `tkit_apply_rule(Name,Pre,Eff)`, which corresponds to `tkit_tis_access` predicate `apply_rule/3`
- `tkit_reset`, which corresponds to `tkit_tis_access` predicate `reset/0`
- `tkit_add_trigger(TC)`, which corresponds to `tkit_tis_access` predicate `add_trigger/1`
- `tkit_clear_triggers`, which corresponds to `tkit_tis_access` predicate `clear_triggers/0`
- `tkit_print_state`, which corresponds to `tkit_tis_access` predicate `print_state/0`

Use and Licenses

TRINDIKIT is available under GPL (Gnu Public License) and it has been downloaded by other research groups (there are 829 downloads by 6/12/2006).

Links

- TRINDIKIT 4 and previous versions can be downloaded from `http://www.ling.gu.se/projekt/trindi/trindikit`

3.4 Grammatical Framework (GF)

3.4.1 Functionality

GF is a framework in which one can build and use multilingual grammars. It comes with a special-purpose functional programming language for writing grammars, a grammar compiler, parser generator, and interactive editor. Moreover, there are standard libraries ("resource grammars"), which help writing GF application grammars.

3.4.2 Runtime Behaviour

GF can be used both as a batch-mode grammar compiler and as an interactive translator ("Multilingual Authoring").

3.4.3 User Definable Knowledge Sources

The user definable knowledge sources are GF grammars and a customizable command option database.

3.4.4 Input/Output Interfaces

GF uses Unix studio and a Java GUI.

3.4.5 Links

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

3.4.6 Latest Updates

GF version 2.4 was released 22 December, with the following main novelties:

- Generation of SLF format for ATK.
- Speech input in the GF interpreter, via on-the-fly SLF generation.
- A notation for transfer rules to complement interlingua-based translation.
- Probabilistic GF grammars.

Also, Resource grammar library v 0.9 has been released, with

- API implemented for 9 languages (6 complete, 3 with some restrictions)
- Multimodal generalization of the resource API, available in 3 languages (English, French, Swedish)

3.5 Prolog API for GF

3.5.1 Functionality

This is a library of Prolog files for parsing Multiple Context-Free Grammars.

There are also utilities for extracting syntax trees and forests from the chart; and for linearization (i.e. generation of strings from syntax trees). The grammars can be multilingual, meaning that this can be used as a simple translation system.

There is also support for Grammatical Framework (GF) grammars, meaning that the library can be used as a Prolog API for parsing, linearization and translation of GF grammars.

3.5.2 Links

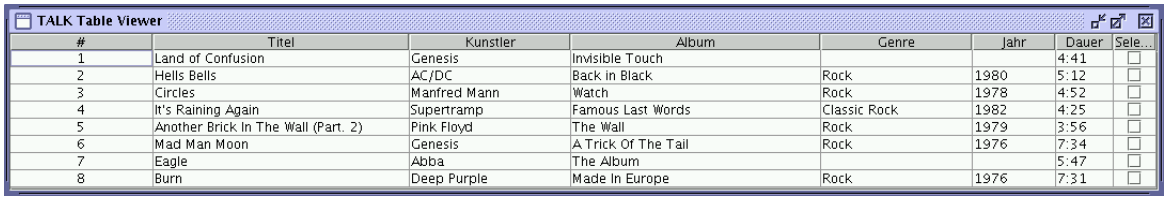
`http://www.cs.chalmers.se/~peb/software.html`

3.6 Table Presenter

The table presentation module from DFKI is one of two generic graphical display modules that have been build in TALK. The module is wrapped as an OAA agent and can display lists of data sets in a number of ways through a simple API that makes it useful in various settings.

One task that arises when accessing databases is the presentation of potentially large sets of results from a query. In the graphics modality, this is usually implemented by presenting a list or table of the results. Graphical user interfaces typically have a single method to present large tables: they layout the entire table in a virtual window which is clipped to the size of the actual window and can be accessed with scrollbars. How this type of presentation task can be enhanced in a multimodal interface is one of the research questions in workpackage 3 and detailed in the description of the Wizard-of-Oz experiments in the status reports.

To support this work, the table presenter can be used to show a set of results either as a list of data, or as a table with headings, modifiable column orders and widths (see figure 3.3 for a table with variable with columns), or instead simply display a text message. See figure 3.4 for examples of all variants. Note also that the last column in the tables contains checkboxes that can be used to select items (e.g., MP3 songs) from the table. We used this feature in the Wizard-of-Oz experiments.



#	Titel	Kunstler	Album	Genre	Jahr	Dauer	Sele...
1	Land of Confusion	Genesis	Invisible Touch			4:41	☐
2	Hells Bells	AC/DC	Back in Black	Rock	1980	5:12	☐
3	Circles	Manfred Mann	Watch	Rock	1978	4:52	☐
4	It's Raining Again	Supertramp	Famous Last Words	Classic Rock	1982	4:25	☐
5	Another Brick In The Wall (Part. 2)	Pink Floyd	The Wall	Rock	1979	3:56	☐
6	Mad Man Moon	Genesis	A Trick Of The Tail	Rock	1976	7:34	☐
7	Eagle	Abba	The Album			5:47	☐
8	Burn	Deep Purple	Made In Europe	Rock	1976	7:31	☐

Figure 3.3: Screenshot from the table presenter module. It shows a table of MP3 songs to choose from.

The basic table presenter component has been the source of a series of follow-up and extended table presenting modules within the WOZ setting itself as well as the SAMMIE MP3 baseline and final showcase

systems of 2005 and 2006 as shown in the following subsections.

3.6.1 Options Presenter

For the Wizard-of-Oz experiments, we needed to present the wizard with a number of possible presentations to choose from. When a list of songs, artists, or titles has been found, we want to find out how much of this information should be presented and in what style. To this end, DFKI has developed a variant of the table presentation agent that can present four different options, allow the wizard to select one of them with a mouse click and send the selected presentation to the user. The screen of the option presenter module then also changes to the second tab, “User Screen” (see the top of figure 3.4) , to mirror the user screen.

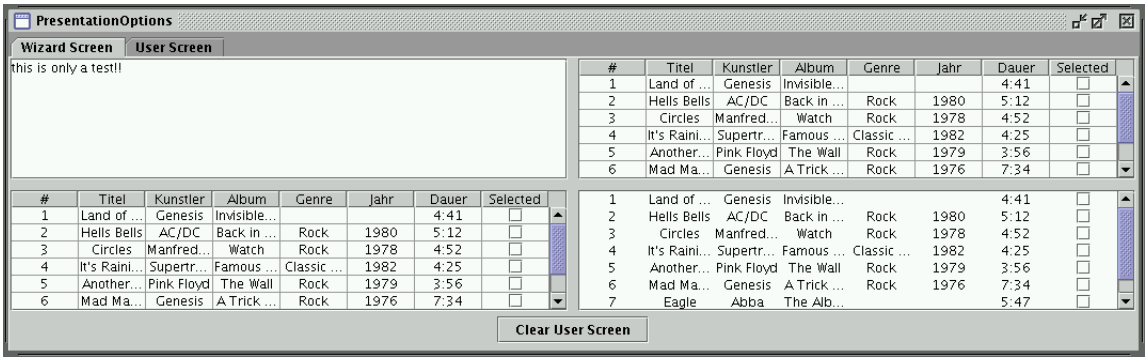


Figure 3.4: Screenshot from the options presenter module. It shows four different views for the wizard to select from: a simple text message, two tables and a list. Note that the list layout is currently being redesigned.

3.6.2 In-car showcase GUI development

In the DFKI/USAAR MP3 in-car baseline system of 2005, the graphical output presentation consists of a combination of several output elements, one of which is a significantly extended and improved version of the basic table presenter of spring 2005. Figure 3.5 presents a screenshot of the complete graphical user interface used in the baseline in-car system.

The table presentation tool of the baseline system distinguishes between marking a line and selecting it explicitely via checkbox. Furthermore, line selections are visualized by the extra checkbox column.

In comparison to the original baseline system version of spring 2005 the most important changes and improvements on the table presenter tool for the final showcase system are the adaptation to the BMW in-car presentation guidelines (concerning, e.g., font color, font size, background color, screen size, number of lines) and the integration of the Ergocommander device control into the system as a simulation of the BMW i-Drive. However, there is no more real distinction between marking and selecting a table line, rather marking corresponds to browsing the selecting line via iDrive turning, and selecting a line via iDrive push immediately triggers the associated system action. In this way, the checkbox column could be omitted without substitution. Thus, we changed the underlying table model such that each individual entry points to a full ontological representation of the graphically presented object. If the user selects an entry by the iDrive or the corresponding keyboard shortcuts, the appropriate ontological representation will then

Figure 3.5: Screenshot DFKI/USAAR in-car baseline system's GUI, a table of artists is shown.

be send to the system's interpretation module. Furthermore, multimodal input including a reference to the table output now simply refer to the highlighted line. There is no longer the need to promptly select that line haptically with the iDrive.

The table visualization has been rounded at the right to indicate the associated sense of rotation of the iDrive. Furthermore, there is a scrollbar that indicates the relation of the viewport to the overall number of table items. Consequently, the table tool can accomplish oaa solvables requesting ontological representations of the currently selected line as well as all objects of the current viewport.

The improved GUI design is shown in figure 3.6. The core characteristics were elaborated and specified at a joint meeting with BMW, Bosch, DFKI and USAAR and then implemented at DFKI for the final in-car showcase.

3.7 Generic Database Module

As a number of systems in TALK are dealing with database access, DFKI has provided a generic MySQL query OAA agent that connects to MySQL-based databases through OAA and provides the results of generic SQL queries which consist of the set of result columns and conditions on some of the columns. The latest version of this OAA agent was released on June 10, 2005 as Version 3.

3.7.1 Solvables

Here is a short description of the solvables:

- `connect( host, port, user, pass, database, Result )`

  - connect to a database via ICL

- `sendGenericDBQuery([column0, column1, ... , columnN-1], [condition0, condition1, ... , conditionM-1], Result)`

  - get N columns from those rows where M conditions hold

Figure 3.6: Screenshot of in-car final showcase system's GUI, main menu.

Arguments are two lists and a variable in which we place the result. The first list indicates the columns we want to retrieve, e.g., *description* or *cuisine*. Note that you can also retrieve all columns by using the wildcard "*".

The N columns of row i are only retrieved when M conditions hold. A valid condition is, e.g., "type=hotel" or "price<30". Note that the concatenation between them is a logical AND. It is also possible to give zero conditions, we indicate that with the empty list, e.g. "[]".

Here are some examples taken from the UEDIN/UCAM tourist information scenario:

`oaa_Solve(sendGenericDBQuery( [id, name], ['type=hotel', 'price<30'], Result ), [] )` would retrieve all hotels which are cheaper than 30 and print their ids and names: `sendGenericDBQuery( [id, name], ['type=hotel', 'price<30'], [ [H1, HOTEL PRIMUS],[H5, ART HOUSE HOTEL], [H6, ALEXANDER HOTEL] ] )`

`oaa_Solve(sendGenericDBQuery( [id, name, 'X', 'Y'], ['type=bar'], Result ), [] )` would retrieve all bars and print their id, name and location.

`oaa_Solve(sendGenericDBQuery( [*], ['type=restaurant'], Result ), [] )` would retrieve all columns of all restaurants.

`oaa_Solve(sendGenericDBQuery( [id, description], [], Result ), [] )` would retrieve id and description of every entity in town

`oaa_Solve(sendGenericDBQuery( [name, room_type], ['type=bar'], Result ), [] )` yields "none" as room_type in the resulting list as a bar offers no rooms to accommodate. Similarly, the cuisine of a hotel would be also "none".

In the context of the DFKI/USAAR final showcase system we have implemented an API for "compiling" database results directly into sets of ontology instances in XML according to predefined result types. This

compilation allows us to map database objects directly into the corresponding ontology objects for further processing within our dialog system.

3.8 iDrive/Ergocommander

BMW has provided software components (and all necessary hardware) to use the ergocommander, a iDrive simulation, in the TALK project. This has enabled the partners to test the in-car systems in a more realistic setting. Currently available is a connection of the input device through the communication bus as used in the car to a serial interface (that can also be acessed through a USB port) and interface software that maps the movements of the input device to keyboard events.

This setup is connected to the GUIs used for the in-car systems which in turn are wrapped in an OAA agent to modularize the setup as much as possible. We were thus able to implement other parts of the input interface, e.g., graphical displays, independently from the actual input device, even simulating the ergocommander with a keyboard.

3.8.1 Final in-car showcase systems

For the final in-car showcase system, the haptic interoperability with the system was extended by (i) the new distinction between short and long iDrive pushes, (ii) the addition of two control buttons at the steering wheel for “Push-to-Talk” and “Push-to-Interrupt” functionality (as described in D5.3), and (iii) the use of the “Menu” button of the iDrive device. For the integration of the system into the car, correspondingly extended EventBridge software for iDrive and steering wheel were provided. On the part of the final showcase system, the new events were realized by means of appropriate key listeners, also to be able to simulate them via keyboard.

3.9 Generic Display Agent

The generic display agent is an OAA agent for displaying several different graphical objects (labels, circles, buttons) on top of a background figure. It has been developed by DFKI in close cooperation with UEDIN/UCAM as it was used in their baseline system to realize parts of the graphical output presentation. The latest version of this agent was released on August 12, 2005 as Version 3.

3.9.1 Solvables

Here is a short description of the solvables:

- `displayAgent_change_Background_image(PathToNewImage, Result)`
  - change the background image
  - example: `oaa_Solve(displayAgent_change_background_image( '../../data/userMap.gif', 'Result'), [])`

- `displayAgent_display_circle(PositionX, PositionY, Radius, RGB_Red, RGB_Green, RGB_Blue, Label, Result)`
  - draw a circle around a point (x,y) with the given radius and the defined RGB-color
    RED : (255,0,0)
    GREEN: (0,255,0)
    BLUE : (0,0,255)
  - example: `oaa_Solve(displayAgent_display_circle(150,100,50,255,0,0, 'myCircle','Result'), [])`
- `displayAgent_remove_circle(Label, Result)`
  - remove the specified circle
  - example: `oaa_Solve(displayAgent_remove_circle('myCircle','Result'), [])`
- `displayAgent_add_Button(PositionX, PositionY, Width, Height, Label, Result)`
  - add a button to the frame the button sends out a solvable request("NameOfTheButton_pressed"), which has to be solved by another agent
  - example: `oaa_Solve(displayAgent_add_button(100,100,50,10, 'MyButton','Result'),[])`
- `displayAgent_remove_Button(Label, Result)`
  - remove the specified button which was added by the addButton request
  - example: `oaa_Solve(displayAgent_remove_button('MyButton','Result'), [])`
- `displayAgent_display_text(PositionX, PositionY, Text, Label, RGB_Red, RGB_Green, RGB_Blue, Result)`
  - display a given text at point (x,y) with a label and a defined RGB-color RED : (255,0,0) GREEN: (0,255,0) BLUE : (0,0,255)
  - example: `oaa_Solve(displayAgent_display_text(100,100,'MyText','MyLabel', 255,0,0, 'Result'), [])`
- `displayAgent_remove_text(Label, Result)`
  - remove the text with the given label which was added by the display_text request
  - example: `oaa_Solve(displayAgent_remove_text('MyLabel', 'Result'), [])`
- `displayAgent_remove_all(ObjectType, Result)`
  - remove all Objects of type "ObjectType". These can be 'Circles', 'Textlabels', or 'Buttons'
  - example: `oaa_Solve(displayAgent_remove_all('Circles', 'Result'), [])`
  - example: `oaa_Solve(displayAgent_remove_all('Textlabels', 'Result'), [])`
  - example: `oaa_Solve(displayAgent_remove_all('Buttons', 'Result'), [])`

3.10 Map Agent

MapAgent is an OAA agent which displays a map showing a graph on top of a background image. The agent has solvables for drawing labelled edges on the map and for getting user clicks on nodes. MapAgent is currently geared towards transport networks and other systems which can be represented by graphs.

3.10.1 Links

MapAgent is currently distributed as part of the tramdemo system.

- Tramdemo system: `http://www.cs.chalmers.se/~bringert/gf/tramdemo.html`
- MapAgent documentation: `http://www.cs.chalmers.se/~bringert/gf/ map-agent.html`

3.10.2 Solvables

- `draw(+Instrs)`: `Instrs` is a string which consists of a number of drawing instructions. Each drawing instruction is terminated by a semicolon. The supported instructions are:
  - `clear` - Clear all old drawings.
  - `drawEdge(label, [node_label, node_label, ...])` - Draw some edges with a common label.
- `clicks(-Clicks)` - Clicks is a chronologically ordered list of clicks that have occurred since the last time this solvable was called. Each click is represented by a struct: `click(Nodes)`, where Nodes is a list of node labels that the click was close to.

3.11 Further General Purpose Modules

This section also contains descriptions of OAA-wrapped general purpose modules of the TALK project that did not change since the publication of version 1 of this deliverable. As they were valuable contributions at earlier stages of the project, we include them as well to ensure the completeness of the documentation in that point.

3.11.1 Keyboard Input Module

For a more specific purpose, namely the Wizard-of-Oz experiments at Saarbrücken, DFKI has developed a module for online transcription of speech that allows the simulation of a speech recognition module. The module in wrapped as an OAA agent and comes in two variants. The first version is intended for a typist that transcribes the user's utterances. The transcribed text is then displayed in a separate window on the wizard's screen. The second version is intended for a typist transcribing the wizard's utterances. The transcribed text is then sent to a speech synthesis system (Mary). This version also allows the typist to mark parts of the utterances as English by enclosing them in parentheses. Note that the experiments are conducted in German, but most song titles and artists are in English. The keyboard input module

then creates appropriate XML markup for the Mary speech synthesis system which can switch between multiple languages.

Another feature of the keyboard input module is spelling correction. To avoid typos, in particular when driving the speech synthesis modules, we have included a spelling correction phase after typing where the typist can choose (by simple keyboard commands to speed up the process) from proposed spelling variants that are taken from a dictionary. This allows us to easily include the correct spellings, e.g., for artists names like “Limp Bizkit”.

The panic button in the top section of the window sends appropriate warning messages to the user or wizard in case the typist is unable to keep up with a fast speaking participant. For example, the wizard who assumes to see actual speech recognition output instead sees the message “Speech recognition error.” when the user typist hits the panic button. Figure 3.7 shows the interface during the spelling correction phase with two options (within the maximal edit-distance of 2) for the word “you”.

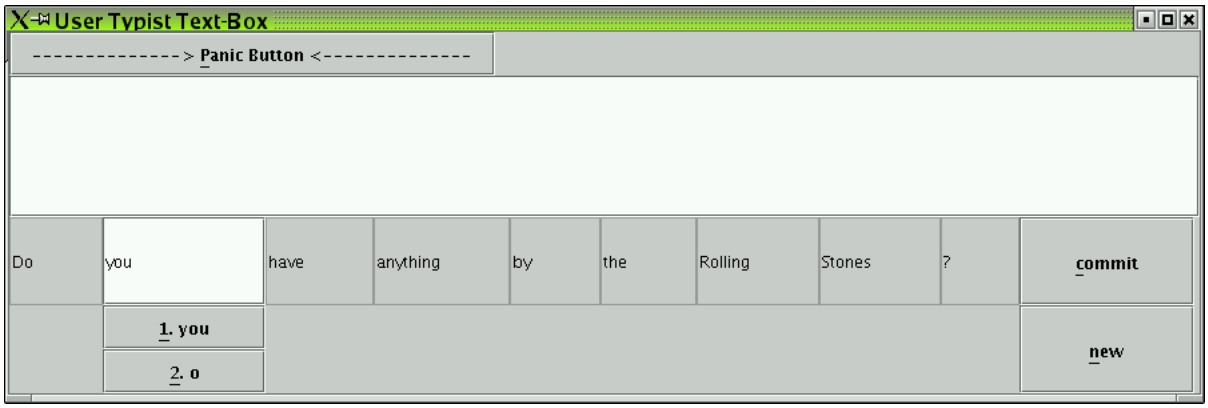


Figure 3.7: Screenshot from the keyboard input module. The screen shows the spelling correction phase after typing is complete. Simple keyboard commands allow the selection of spelling variants found in the dictionary.

3.11.2 Pen Input Module

In systems that provide a mouse or pen input modality, typical applications are drawing (e.g., a route on a map), selecting (e.g., clicking on a song title), and gesturing (e.g., encircling a location on a map). To provide a generic access to the device data, DFKI is currently developing a generic pen input module that can capture pen (or mouse) input by collecting a stream of coordinates from the input device. These can then be displayed in a window which will have an arbitrary background image, e.g., a map. The coordinates are also available for output, e.g., in the InkML markup language. In the current prototype, coordinates are printed in the console.

Sets of coordinates will be collected as strokes that can then be modified, e.g., erased, individually.

Development on this component has been stopped in mid 2005, because then it turned out that capturing of pen input strokes is definitely not used in the TALK systems. The corresponding input functionalites are replaced by mouse and iDrive/Ergocommander input or are realized by the generic display agent.

Chapter 4

Individual Modules with OAA Middleware Wrappers

This chapter describes indivudual OAA-wrapped modules or components of TALK showcase systems. The relevant work on those components within the TALK project ranges from just OAA-wrapping an off-the-shelf component via extending an already existing module to newly implement a specific modality.

4.1 ATK Real-Time Speech Recognition

ATK recognises speech from a live stream or set of pre-recorded files. It can use N-grams language models and/or recognition grammar networks, switching resources on the fly. N–best recognition is supported and all incoming speech can be saved as time/stamped wav files.

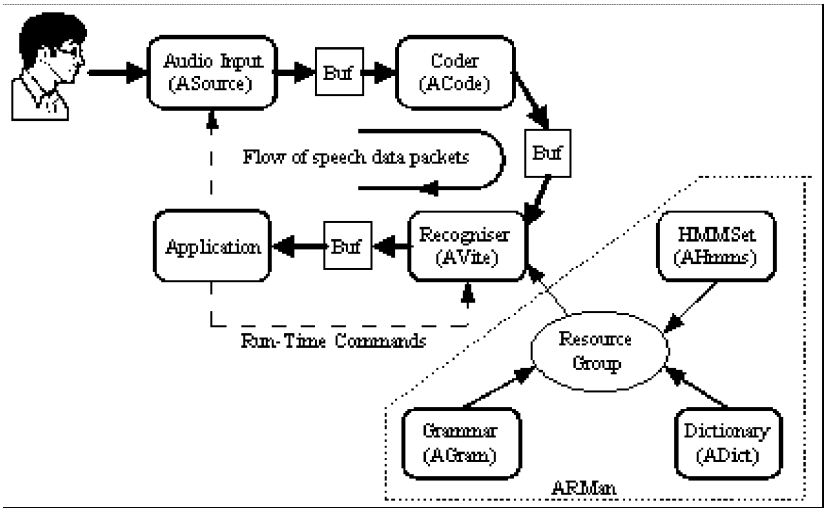


Figure 4.1: Sketch of the ATK internal architecture.

Figure 4.1 provides a sketch of the ATK architecture. ATK was developed by the University of Cambridge. It defines a series of classes - components, buffers and packets. The components are connected together

using buffers and communicate by sending packets to each other. The components are e.g.: the speech source, the audio coder and the recogniser. Packets could be speech waveforms, coded speech parameters, command signals or recognised text strings.

Output is in the form of text strings/packets. ATK exists as a set of C++ classes built on top of the HTK C libraries. Linux and Windows are supported. A wrapper for OAA has been completed by UEDIN.

ATK requires a grammar network and/or a language model, as well as a pronunciation dictionary and a set of HTK-compatible acoustic models. It was used in the TALK dialogue systems of [11, 10, 16].

Links

- `http://htk.eng.cam.ac.uk/`

4.2 OWL-based Knowledge Manager

USE has developed an agent that provides a generic interface for making RDQL queries to an OWL ontology. The fact that OWL is based on RDF, and therefore structured as triplets Subject–Property–Object, is used to provide an interface where the client specifies the Subject and gives restrictions over its Range or, alternatively, specifies the Object and gives restrictions over the Domain.

Remarkably, the Ontology hierarchy is hidden by the agent which uses the OWL reasoner to provide an accurate answer no matter how the Ontology graph is built.

4.2.1 Solvables

- setProperty(Subject, Property, ObjectList)
  - Subject: Specify the subject (as an individual).
  - Property: Property name
  - ObjectList: List of new values for this property of that subject.
- getIndividualsFromRange(L_ind, L_prop, Result)
  - L_ind: List of instanced objects (individuals).
  - L_prop: List of properties whose value we are interested in.
  - Result: List of results of this query.
- getIndividualsFromDomain(NameClass, Pos_filter, Neg_filter)
  - NameClass: Name of the class from where we want to get the individuals.
  - Pos_filter: List of pairs [nameProperty, value], properties which the object must be in accordance with.
  - Neg_filter: List of pairs [nameProperty, value], properties which the object must NOT be in accordance with

4.3 Atlas ASR Module

This agents wraps the Atlas ASR API providing the same interface as the Nuance's Agent. This module is particulary interesting in some scenarios because Atlas Language Models are optimized for the Spanish official languages (Spanish, Catalonian, Basque and Galician).

It can work in synchronous mode, where the agent is given the control by a "GetSpeech" call, or asynchronously, where the agent is always recognizing and putting the results ("PutSpeech") in the multimodal input pool.

4.3.1 Solvables

- getSpeech: No argument. It returns as a string the utterance with highest confidence from the recognition hyhpothesis.

4.4 Atlas TTS Module

This agents wraps the Atlas ASR API. As previously mentioned, Atlas products are optimized for Spanish official languages.

Besides the text string to be synthesized, this agent also accepts the working language allowing multilinguallity at execution time.

4.4.1 Solvables

- sayText(Text): Makes the synthesizer pronounce a text. No outputs.
  - Text: Text string to be synthesized
- setLanguage(Language): Makes the synthesizer switch from one language to another. No outputs.
  - Language: New language configuration ('spanish', 'english' or 'catalan'). By default this agent is configured to use Spanish.
- setVoicePitch(Pitch): Allows the fundamental frequency (tone or pitch) to be modified, expressed in semitones (st). It doesn't have any output.
  - Pitch: Integer Value. User scale 0-100
- setVoiceSpeed(Speed): Allows the speaking rate to be modified, expressed in words per minute (wm). It doesn't have any output.
  - Speed: Integer Value. Words per minute

4.5 Multimodal Input Pool Module

This agent provides an asynchrony layer between the User Interface modules and the Dialogue Manager. From an implementation point of view, this is just a FIFO (First In First Out) queue that stores all the inputs (Push: PutInput) until another agent asks for it (Pop: GetInput).

4.5.1 Solvables

- PutInput(modality, utterance, time_init, time_end, confidence): Called by the input modality agents when a new user input is received.
  - Modality: Communication channel, i.e., “voice”, “click”, “gesture”
  - Utterance: User Input, i.e., sentence said by the user, icon clicked, etc.
  - Time_init: Time where the utterance began.
  - Time_end: Time where the utterance ended.
  - Confidence: Recognition rate (at the time being only used by the ASR agents)
- GetInput. No arguments. Returns the same five parameters as the previous solvable, in the same order.
- CheckInput. Checks if there is any element in the queue.

4.6 Microsoft Animated Agent Module

This agent wraps the Microsoft Animated Agent API, providing the same interface as the Atlas TTS agent (sayText, setLanguage, setPitch, setSpeed).
The Microsoft Animated Agent Module was used for the USE WOz experiments with the “Merlin” character.

4.7 Loquendo Animated Agent Module

This agent wraps the Loquendo API, providing the same interface as the other TTS agents (sayText, setLanguage, setPitch, setSpeed). But it also has a face whose lips and facial expressions are synchronized with the syntectic voice, and the ability to express different moods and play some predefined motions.
Besides the same solvables as the other TTS agents, this one offers the following ones:

4.7.1 Solvables

- setExpression(ExpressionName): Sets the talking head expression, simulating the new mood of the character. Replace the previous mood. It doesn’t have any output.
  - ExpressionName. Identifier of the expression. Currently it only accept these moods:

    - Anger
    - Doubt
    - Fear
    - Happiness
    - Idle
    - Sadness
    - Surprise

- playMotion(MotionName): Makes the talking head play a motion, like nod or shake. It doesn't have any output.
  - MotionName. Identifier of the motion to be played. Currently it only accept these motions:
    - nod
    - shake

- wakeUp: Makes talking head wakeUp. If it is not asleep, the solvable doesn't have any effect. It doesn't have any output. No parameters either.

- sleep: Makes talking head fall asleep. If it is already asleep, the solvable doesn't have any effect. It doesn't have any output. No parameters either.

4.8 House Setup Manager

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.) are 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.

4.8.1 Solvables

- ZoomIn(Location): Zooms in the room specified by the parameter "Location".
  - Location: Room to be zoomed.

- ZoomOut: Loads the general view of the House. No parameters.

- hsUpdateDeviceState(Label, State): This solvable is called whenever there is a change in a particular device. The HomeSetup agent updates the icon of the device and the house representation (e.g., if the kitchen light is switched on, its icon is updated and the kitchen is illuminated).
  - Label: Device identifier.
  - State: This is a numerical value, "0" means "off" (or down, when the device is a blind) and "100" means "on" (or "up", when referring to a light). Intermediate values are allowed only when the device is classified as a dimmer in the ontology.

4.9 Device Manager

This agent acts as a bridge between the system and the real devices. Whenever a command is received (i.e., ”switch on Lamp_1”), the agent translates it to the x10 equivalent instruction and updates the Ontoloty and the House Layout.

4.9.1 Solvables

- ExecuteAction(command, device_address): This agent offers a single solvable, from where all the command requests are received.
  - Command: Action to be executed: on, off, dim and bright.
  - device_address: X10 address of the device.

4.10 USE WOz Experiments Specific OAA Agents

4.10.1 Wizard Agents

1. **Wizard Helper**

   This is basically a control panel that enables the wizard to:

   - Talk to the user. The panel is connected to a TTS running on the user’s computer. The wizard can either choose among a number of possible sentences (previously determined according to the possible actions of the user) or type an alternative answer if the user’s behavior differs from what had been foreseen.
   - Remotely play audio and video files (to simulate the camera and telephone).

   The Wizard Helper panel can switch the experimental setting from one language to another by just picking up a flag.

2. **Wizard Device Manager**

   This agent connects the wizard computer with the physical devices and with the user’s Home Setup. When the Wizard clicks on the “kitchen light - on” button, this agent sends an X10 message to the kitchen light and also updates the user’s Home Setup.

4.10.2 User Agents

1. **Home Setup**

   This is a modified version of the actual system agent that displays the current setting of the house and its devices. The user may use the mouse or pen to click on the devices. When the device is clicked, it blinks (so that the wizard can see it with his remote screen) and sends a log message to the Log Manager. The Home Setup is linked to the Device Manager, so as to ensure its immediate update.

2. **Telephone Simulator**

   This is the telephone terminal access icon on which the user can click. It blinks when clicked on and sends a message to the Log Manager as the rest of the Home Setup devices.

3. **Log Manager**

   This agent keeps record of all the user-wizard interactions during the experiment. It includes the information sent by the GUI Agents (Home Setup and Telephone Simulator) and the voice Agents (TTS and ASR Manager). This is the information saved in execution time during the experiment. The logging is therefore focused on low-level information, and especially on the time at which each utterance occurs. The following table resumes the information logged:

Modality	Information Logged
GUI Input	Icon clicked Time
GUI Output	Message Time
Voice Input	General: Recognizer, Grammar, Language. Hypothesis level: Sentence, Score, Time Init, Time End. Word Level: Word, Score, Time Init, Time End.
Voice Output	Message Time

   All this information is logged following the schema proposed by the W3C recommendations from Emma (www.w3.org/TR/emma/, still a working draft), with very few modifications.

4. **Video Client**

   This specific agent plays media files (video and audio) where asked by the Wizard Helper. The files are played in different execution threads so that they can be stopped at any time.

4.11 iCal Agent

4.11.1 Functionality

This is an OAA agent that parses and generates calendar files of the iCal format. At UGOT, it is used to connect AgendaTALK with calendar applications using the iCal format.

The iCal agent offers two basic services: searching and adding iCal items.

4.11.2 Links

- iCal Agent: `http://www.ling.gu.se/projekt/talk/software/`
- iCAL: `http://www.apple.com/ical/`

4.11.3 Solvables

- `add(+Type, +Summary, +StartDate)` - Add event of type `Type`, which can be either of the following:

  - `todo`
  - `alarm` `Summary` is a string describing the event. `StartDate`, and dates generally, takes the form *YYYYMMDDTHHMM*, e.g. `20041215T1530`.

- `add(+Type, +Summary, +StartDate, +StopDate)` - As `add/2` but with an extra argument indicating stop date.
- `entry(Entry)` - Search the database for entry `Entry`. Entries have one of the the following formats:
  - `entry(event(?Summary, ?Id, startDate(?StartDate), endDate(?EndDate)))`
  - `entry(todo(?Summary, ?Id, startDate(?StartDate)))`

  where `Summary` is again a string describing the event, `Id` is an integer ID assigned to the event by the iCal agent, and `StartDate` and `EndDate` are dates, as described above.
- `delete(+Id)` - Delete an entry from the database. `Id` is the ID number of the entry to delete (see `entry/1`).
- `buildCal(+CalendarFileName)` - Convert current database entries to text strings, and write them to an `*.ics` file indicated by the string argument `CalendarFileName`.
- `buildDb(+CalendarFileName)` - Parse the calendar file indicated by string argument `CalendarFileName` and make a database of the corresponding entries.

4.12 jlGui Wrapper Agent

4.12.1 Functionality

The wrapper connects jlGui, an off-the-shelf audio player, to OAA. JlGui can handle WAV, AU, AIFF, SPEEX, MP3 and OGG Vorbis file formats as well as the streaming variety of the latter two. It is a graphical Java interface with playlist and equaliser capabilities which is built around a core basic audio player, a Java package that offers basic audioplayer functions. These include opening media, jumping to a certain point in the media (“FF”, “REW”), playing, pausing, resuming and halting playback as well as manipulating the volume of the output and the balance between speakers.

The agent does not make use of the graphical interface but communicates with the BasicPlayer class that is the foundation of jlGui. Basically this means that the commands available through the BasicPlayer class used in jlGui are implemented in the agent as well.

The wrapper thus incorporates the basic audioplayer package as well as the playlist functions. It does not, however, allow access to the graphical interface or the equaliser functions as these were tightly woven into the GUI and thus not easily manipulated.

4.12.2 Links

- jlGui Wrapper Agent: `http://www.ling.gu.se/projekt/talk/software/`
- jlGui: `http://www.javazoom.net/`

4.12.3 Solvables

- `addFile(+File,?Item)` - Appends the audio file `File` to the playlist. `Item` is the item corresponding to the added file.
- `addFile(+File, +Index, ?Item)` - Appends the file `File` to the playlist position indicated by `Index`. `Item` is the item corresponding to the added file.
- `addURL(+Url, ?Item)` - Appends the URL `Url` to the playlist. `Item` is the item corresponding to the added URL.
- `addURL(+Url, +Index, ?Item` - Adds the URL `Url` to the playlist position indicated by `Index`. `Item` is the item corresponding to the added URL.
- `clearPlaylist` - Clears the playlist
- `getIndex(?Index)` - Returns `Index`, the index of the current item in the playlist
- `getPlaylistSize(?Size)` - Returns `Size`, the size of the loaded playlist
- `makePlaylist(+Locations,?Items)` - Makes a playlist out of `Locations`, which is a list of filenames and/or URLs. `Items` is the list of items in the playlist.
- `openPlaylist(+File,Items)` - Opens the playlist saved as file `File`. `Items` is the list of items in the playlist.
- `pausePlayer` - Pauses the playback .
- `playNext` - Plays the next item.
- `playPlayer` - Starts the playback .
- `playPrevious` - Plays the previous item.
- `removeCurrent` - Removes the current item from the playlist.
- `removeIndex(?Index)` - Removes the item at index `Index` from the playlist.
- `resumePlayer` - Resumes playback after pausing.
- `stopPlayer` - Stops playback. playback
- `seekPlayer(+Byte)` - Starts playback of the current item from the frame closest to the given byte `Byte`. Does not work with streams.
- `seekPlayerPercent(+Percent)` - Starts playback of the current item from the frame indicated by `Percent`. Does not work with streams.

- `setGain(+Level)` - Changes the signal boost, volume control. `Level` is a float value between 0.0 and 1.0.
- `setPan(+Pan)` - Controls the balance between speakers. `Pan` is a float value between -1.0 and 1.0.
- `showList(?List)` - Retrieves the current playlist `List`, as a list of items.
- `saveList(+File)` - Saves the current playlist to file `File`.
- `shuffleList(?List)` - Shuffles the current playlist. `List` is the new playlist as a list of items.
- `whatIsPlaying(?Item)` - `Item` is the currently playing item.

The structure of an item is as follows:
`item(FormattedName,Length,Location,Info)` where

- `FormattedName` is an easy-to-read description of the item, consisting of e.g. the song title and the artist
- `Length` is the length of the item in seconds
- `Location` is the location of the item, i.e. a filename or an URL
- `Info` is a list which elements are:
  - `title(Title)`
  - `artist(Artist)`
  - `album(Album)`
  - `genre(Genre)`
  - `year(Year)`
  - `track(Track)`
  - `playTime(PT)`
  - `samplingRate(SR)`
  - `bitRate(BR)`

  All elements in the `Info` list are optional.

4.13 ffmpeg Wrapper Agent

4.13.1 Functionality

This OAA agent is written in Java. It is a wrapper for ffmpeg which can capture audio and video from a television card. It runs on a Linux machine with a TV card.
The agent can schedule recordings of TV programs, and delete scheduled recordings. Recordings eventually show up on password-protected web page. The scheduler is designed for several users (but only one

at a time). Each user manages her own recordings. The agent keeps track of one text file for users and passwords, and one text file for available channels.

The actual recording is done by the program ffmpeg (`http://ffmpeg.sourceforge.net`) which is started by the Linux `cron` utility at a specific time. Thus the list of scheduled recordings are kept in the Linux crontab.

4.13.2 Links

- ffmpeg Wrapper Agent: `http://www.ling.gu.se/projekt/talk/software/`
- ffmpeg: `http://ffmpeg.sourceforge.net/index.php`

4.13.3 Solvables

The structure of an item is as follows:
`item(FormattedName,Length,Location,Info)` where

- `FormattedName` is an easy-to-read description of the item, consisting of e.g. the song title and the artist
- `Length` is the length of the item in seconds
- `Location` is the location of the item, i.e. a filename or an URL
- `Info` is a list which elements are:
  - `title(Title)`
  - `artist(Artist)`
  - `album(Album)`
  - `genre(Genre)`
  - `year(Year)`
  - `track(Track)`
  - `playTime(PT)`
  - `samplingRate(SR)`
  - `bitRate(BR)`

  All elements in the `Info` list are optional.

- `checkUser(+User, +Password, -Result)`
  - Result is `true` if user `User` has password `Password`, else `false`.
- `addJob(+Usr, +Channel, +Start, +Stop, -Result)`
  - Adds a “rec job” (i.e. a scheduled recording) for user `Usr`.

  - Starts recording channel `Channel` at time `Start`.
  - Ends recording at time `Stop`.
  - Result can be either of
    * `ok` (the VCR agent successfully added the rec job)
    * `time_conflict` (there is an overlapping scheduled recording)
    * `non_existing_user` (the user does not exist)
    * `non_existing_channel` (the channel does not exist)
    * `old_start_time` (the start time has already passed)

- `removeJob(+Usr, +JobId)`

  The structure of an item is as follows:

  `item(FormattedName,Length,Location,Info)` where
    * `FormattedName` is an easy-to-read description of the item, consisting of e.g. the song title and the artist
    * `Length` is the length of the item in seconds
    * `Location` is the location of the item, i.e. a filename or an URL
    * `Info` is a list which elements are:
      · `title(Title)`
      · `artist(Artist)`
      · `album(Album)`
      · `genre(Genre)`
      · `year(Year)`
      · `track(Track)`
      · `playTime(PT)`
      · `samplingRate(SR)`
      · `bitRate(BR)`

    All elements in the `Info` list are optional.
  - Deletes the rec job with id `JobId` for user `Usr`.

- `existsRecJobWithId(+Id)`
  - Checks if a rec job with id `Id` exists.

- `getRecJobs(+Usr, -Jobs)`
  - `Jobs` is a list of the planned rec jobs for user Usr.
  - The structure of a rec job is `recJob(Id, Usr, Channel, Start, Stop)`

- `validRecJob(+Usr, +Channel, Start, +Stop, -Result)`
  - Like addJob, but only checks if the job is valid

- `getChannels(-Channels)`

  - Returns a list of the available channels.

- `validChannel(+Channel)`
  - Checks that `Channel` is a valid channel (i.e. is defined in the channels text file).

4.14 Festival Wrapper Agent

4.14.1 Functionality

This agent connects Festival TTS to OAA. It is written in Java. The classes handle input streams to, and audio output streams from, a Festival server. The output streams are made into audio streams by using Java's javax.sound.sampled API.
Note that a separate OAA agent for Festival has been implemented by UEDIN (in the C programming language). See also the synthesis agents in sections 4.14 and 4.26.

4.14.2 Links

- Festival Wrapper Agent: `http://www.ling.gu.se/projekt/talk/software/`
- Festival: `http://www.cstr.ed.ac.uk/projects/festival/`

4.14.3 Solvables

- `sayTxt(+String)` - Send `String` to Festival, which will produce streaming audio.
- `readTxtFile(+TextFile)` - Send the contents of `TextFile` to Festival. The path is either relative from the directory where the Festival agent application is started, or absolute.
- `saveCmdToWave(+String)` - As `SayTxt/1`, but in addition the resulting audio file is saved in the directory where the Festival agent was started. The file is given a default name (`output`*N*`.wav`), where *N* is the current number of files.
- `saveCmdToWave(+String, +AudioFile)` - As `saveCmdToWave/1`, but takes an extra argument `AudioFile` (a string) indicating the name of the saved file. The file will be saved in the directory where the agent is started. If a file with the same name already exists, it will be overwritten.
- `saveCmdToWave(+String, +AudioFile, +Directory)` - As `saveCmdToWave/2`, but also takes a directory name where the audio file will be saved. The path to the directory can be relative(from the directory where the agent is started) or absolute.
- `saveFileToWave(+TextFile)` - As `saveCmdToWave/1` but takes a filename instead of a string.
- `saveFileToWave(+TextFile, +AudioFile)` - As `saveCmdToWave/2` but takes a filename instead of a string.
- `saveFileToWave(+TextFile, +AudioFile, +Directory)` - As `saveCmdToWave/3` but takes a filename instead of a string.

- `changeVoice(+VoiceName)` : Change voice. Festival includes two American male voices and one Castillian Spanish male:
  - `kal_diphone` (default)
  - `ked_diphone`
  - `el_diphone` (Castilian Spanish)
- `playWave(+AudioFile)` - Play an existing audio file. There are three types of arguments you can use as audio file name:
  - `"latest"` - This will play the audio file that was last saved.
  - A path to the chosen audio file.
  - A number *N* corresponding to the number of a file that was saved with a default name, i.e. `output`*N*`.wav`.
- `stopWave` - Stop the audio output from Festival.

4.15 FreeTTS Wrapper Agent

4.15.1 Functionality

The FreeTTS Wrapper Agent connects FreeTTS to the OAA framework. It is possible to synthesize strings and the contents of urls and files. It is also possible to change the voice and to use MBROLA voices.

4.15.2 Links

- FreeTTS Wrapper Agent: `http://www.ling.gu.se/projekt/talk/software/`
- FreeTTS 1.2beta: `http://freetts.sourceforge.net/docs/`
- MBROLA: `http://tcts.fpms.ac.be/synthesis/mbrola.html`

4.15.3 Solvables

- `speakText(+Text)` - Synthesize the string `Text` into audio output.
- `speakFile(+FilePath)` - Synthesize the content of the file pointed to by `FilePath`
- `speakUrl(+URL)` - Synthesize the content of the URL `URL`
- `setMbrolaBase(+MbrolaPath)` - Set the base for MBROLA voices to `MbrolaPath`.
- `changeVoice(+NewVoice)` - Change the voice into `NewVoice`.

4.16 Sphinx 4 Wrapper Agent

4.16.1 Functionality

This wrapper is an interface to the Sphinx-4 recognizer system. It is a general application that can configure any Sphinx4 configuration file and perform some methods for recognition and make changes to the configuration.

4.16.2 Links

- Sphinx 4 Wrapper Agent: `http://www.ling.gu.se/projekt/talk/software/`
- CMU Sphinx 4: `http://cmusphinx.sourceforge.net/sphinx4/`

4.16.3 Solvables

- `recognize(-ReturnedResult)` - Does speech recognition and returns the string of words recognized. Sample result: `simpleResult('good morning philip')`
- `recognizeNBest(+MaxSizeOfNBestList, -ReturnedResult)` - Takes an integer `MaxSizeOfNBestList` as input, giving the maximum number of tokens to be returned as an n-best list. The solvable returns the "best token" with a list of results, and a list of tokens with results (where the best token is included as the first item of the list) This solvable gives back result in the following form:

```
result(
           bestToken(
                token('good morning philip',
                [totalScore(-1.2267607E7),
                languageScore(0.0)])),
           n_bestList([token('good morning philip',
                [totalScore(-1.3278091E7),
                     languageScore(0.0)]),
                     token('good morning philip',
                [totalScore(-1.2267607E7),
                     languageScore(0.0)])]))
```

- `changeRecognizer(+NameOfPreAllocatedRecognizer)` - Changes the current recognizer to another, already allocated, recognizer. This solvable requires the different recognizers to be defined with different names in the configuration file, and also that their names are given as input to the SphinxAgent so they can be allocated at start-up.
- `changeGrammar(+LinguistComponent, +GramComponent, +GramLocation, +GramName)` - Deallocates the current grammar and sets location- and name-properties to a new grammar component.

- `changeDictionary(+LinguistComponent, +DictComponent, +DictPath)` - Deallocates the current dictionary and sets path properties to a new dictionary component.
- `changeGramDict(+LinguistComp, +GramComponent, +GramLoc, +GramName, +DictComponent,+DictPath)` - Combines changeDictionary and changeGrammar.
- `changeLM(+LanguageModelName, +LanguageModelLocation)` - Deallocates the current language model component and sets the location to the new value.
- `saveConfig(+FileName)` - Saves the current configuration to a new configuration file.

4.17 NuanceWrapper Agent

4.17.1 Functionality

NuanceWrapper is an OAA agent which serves as a wrapper to the Nuance SpeechChannel java API for speech recognition and speech synthesis. NuanceWrapper also covers some additional functionality, such as compilation of dynamic grammars, and retrieval of recorded audio files over OAA. NuanceWrapper is designed to be used as a resource called by other OAA agents, and does not call any other OAA agents except when registering to the OAA facilitator.

Some of the Nuance methods generate events during the execution of the method. Since NuanceWrapper is designed to be a passive agent these events are not published directly to the OAA community. Instead the caller agent must get them explicitly by calling the solvable `nscGetEvent`. The solvable `nscSetGenerateEventOfType` can be used to determine what events should be generated.

4.17.2 Links

http://www.ling.gu.se/projekt/talk/software/wrappers.html

4.17.3 Solvables

- `nscAnswerCall` – Answers an incoming call.
- `nscAppendTTS(+Text)` – Appends `Text` to the TTS queue.
- `nscClearAudioLog+Type` – Clears the log of type `Type` of recorded audio files. Currently the only admitted value of `Type` is `asr`
- `nscClose` – Closes the NuanceSpeechChannel.
- `nscCompileGslToNGO(+Grammar, +MasterPackage, ?PathToNGO/3` – Compiles `Grammar` to a Nuance Grammar Object.
- `nscCreate(+Package, +Parameters)` – Calls the NuanceSpeechChannel constructor
- `nscCurrentMasterPackage(?MasterPackage)` – Queries the name of the current Nuance master package.

- `nscCurrentState(?State)` – Queries the current state of NuanceWrapper. Different solvables are active in different states.
- `nscDefaultParameters(?ListOfParameters)` – Queries the current set of default Nuance parameters of NuanceWrapper and their associated values.
- `nscGetAllGrammars(?ListOfGrammars)` – Returns a list of the names of the current top level grammars.
- `nscGetAudioLog(+Type,?LogFilesList)` – Returns logged audio files.
- `nscGetEvent(+Type,?Event)` – Returns the first event of type `Type` on the NuanceWrapper event queue.
- `nscGetParameter(+Name,?Value)` – Retrieves the `NuanceSpeechChannel` value of a Nuance parameter.
- `nscGrammarDir(?Path)` – Returns the directory where NuanceWrapper looks for recognition packages.
- `nscGslCompiledToNGO(?Grammar, ?MasterPackage, ?PathToNGO)` – Returns true if `Grammar` has been compiled into the Nuance grammar object `PathToNGO` using master package `MasterPackage`.
- `nscHangup` – Hangs up an ongoing telephony call.
- `nscPlay(+Interruptable)` – Requests that NuanceWrapper should start a thread that plays all prompts previously appended by calling `nscAppendTTS/1`.
- `nscPlayAndRecognize+Context` – Requests that NuanceWrapper should start a thread that plays all appended prompts and performs speech recognition using context `Context`.
- `nscRecognizeFile(+Filename,+Context)` – Requests that NuanceWrapper should start a thread that performs speech recognition on the endpointed audio file `Filename` using context `Context`.
- `nscSetDefaultParameters(+ListOfParameters)` – Sets the default Nuance parameters of NuanceWrapper.
- `nscSetGenerateEventOfType(+Type,+YN)` – Sets whether NuanceWrapper should generate a NuanceWrapper event of a certain type.
- `nscSetGrammarDir(+Path)` – Sets the directory where NuanceWrapper looks for recognition packages.
- `nscSetParameter(+Name,+Value)` – Sets the value of a Nuance parameter.
- `nscShouldGenerateEventOfType(?Type)` – Checks whether events of type `Type` will be generated.
- `nscWaitForCall(?CallInfo)` – Waits for an incoming call and returns information about the caller, when a call has been detected.

4.18 SQL Prolog API

This SQL interface written in Prolog connects GoDiS with a MySQL database. It is a general database interface which in the TALK project is used for the calendar application AgendaTALK to be able to alter the graphical calendar application's database. However, the database interface is not application-specific and could be use for any GoDiS application having a MySQL database.

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

4.18.1 Links

`http://wiki.ling.gu.se/bin/view/TALK/TalkSoftwareLibrary`

4.18.2 Prolog API

The SQL interface converts Prolog commands and associated Prolog lists into SQL commands. The following commands are possible:

- `selectDB( +Table, +Query, +Values,-Answer )` SELECT Query FROM Table WHERE Values;
- `selectDBOrderBy(+Table, +Query, +Values, +Order,-Answer)` SELECT Query FROM Table WHERE Values ORDER BY Order;
- `insertDB( +Table, +Values, -Answer )` INSERT INTO Table VALUES Values;
- `deleteSQL( +Table, +Values, -Answer )` DELETE FROM Table WHERE Values;
- `countDB(+Table,+Column, +Values, -Answer )` SELECT COUNT Column FROM Table WHERE Values;
- `updateSQL(+Table,+Field,+NewValue,+Values,-Answer)` UPDATE Table SET Field=NewValue WHERE Values;
- `updateALL(+Table,+Field,+NewValue,-Answer)` UPDATE Table SET Field=NewValue;

The database interface takes the Prolog lists and constructs SQL commands from these, then calls the MySQL database to pose the SQL-query and finally constructs an appropriate answer in return based on the resulting SQL answer message. As an example the `selectDB` Prolog predicate with associated values will be reconstructed into the SQL command shown above. The term Values are the value conditions of a SQL search such as 'X = Y','X LIKE Y%' or 'X LIKE %Y' represented in Prolog as a list of conditions.

4.19 Java GF Agent

4.19.1 Functionality

The GF Agent is written in Java and makes it possible to use GF grammars with OAA to parse and generate (linearize) given a GF grammar.

4.19.2 Links

- GF Agent: `http://www.cs.chalmers.se/~bringert/gf/gf-oaa.html`
- GF: `http://www.cs.chalmers.se/~aarne/GF/`

4.19.3 Solvables

- `parse(+Grammar, ?Lang, +Text, ?Tree)`
  - `Grammar` - The name of the grammar. This is the value of the name parameter in the properties file.
  - `Lang` - The name of the concrete syntax that should be used for parsing. If Lang is not instantiated, the parser will try all available languages in the given grammar, and return results for each language that the text can be parsed in.
  - `Text` - The text to parse. Must be instantiated.
  - `Tree` - The parse tree. Normally not instantiated. The parse tree from the parser is unified with this value. Parse trees are represented as ICL structs.
- `linearize(+Grammar, ?Lang, +Tree, ?Text)`
  - `Grammar` - The name of the grammar. This is the value of the name parameter in the properties file.
  - `Lang` - The name of the concrete syntax that should be used for linearization. If Lang is not instantiated, linearizations for all available languages in the given grammar will be returned.
  - `Tree` - The abstract syntax tree to linearize. Must be instantiated.
  - `Text` - The linearization of the given tree.
- `translate(+Grammar, ?FromLang, +Input, ?ToLang, ?Output)`, where
  - `Grammar` - The name of the grammar. This is the value of the name parameter in the properties file.
  - `FromLang` - The name of the concrete syntax that should be used for parsing the input text. If Lang is not instantiated, all available languages in the given grammar will be tried.
  - `Input` - The input text. Must be instantiated.
  - `ToLang` - The name of the concrete syntax that should be used for linearizing the output.

  - `Output` The output text. Normally not instantiated.

- `list_grammars(?Grammars)`
  - `Grammars` - The list of available grammars.
- `list_languages(?Grammar, -InputLangs, -OutputLangs)`
  - `Grammar` - The grammar to get the languages for. If uninstantiated, there will be one answer for each available grammar
  - `InputLangs` - A list of the available input languages for the grammar.
  - `OutputLangs` - A list of the available output languages for the grammar.

4.20 Borg Agent

4.20.1 Functionality

The Borg Agent is an OAA agent which serves as a wrapper to Borg, where Borg is a calendar and task tracker application written in Java and hosted on SourceForge[1]. The user interface to Borg has a month-at-a-time view which displays the chosen month in the form of a rectangular grid of days. Appointments and todo's are listed in this view on their respective days. By clicking on a day a user can open the appointment editor and add, remove or update appointments. Our OAA agent enables basic functions in Borg to be controlled via OAA, such as opening and closing Borg, and also enables advanced control of the calendar interface, such as highlighting using different colours.

4.20.2 Links

http://www.ling.gu.se/projekt/talk/software/borg.html

4.20.3 Solvables

- `start_calendar` – Start or restart the calendar interface.
- `close_calendar` – Close the calendar interface.
- `change_language(`*Lang*`)` – Change the language used by the calendar interface and restart the calendar.
- `update` – Sync the calendar with the database.
- `next_month` – Display the next month.
- `previous_month` – Display the previous month.
- `goto_date(`*Date*`)` – Display the month containing *Date*.

[1] `http://borg-calendar.sourceforge.net/`

- `set_default_colors` – Remove all highlighting.
- `changecolor_text(`*Color*`,` *Date*`,` *TextString*`)` – Mark *TextString* on *Date* with *Color*.
- `changecolor_text(`*Color*`,` *Date*`,` *Time*`,` *TextString*`)` – Mark *TextString* on *Date* at *Time* with *Color*. Use this to distinguish several appointments on the same date that contain the same text.
- `changecolor_time(`*Color*`,` *Date*`,` *Time*`)` – Mark the *Time* label on *Date* with *Color*.
- `changecolor_digit(`*Color*`,` *Date*`,` *Time*`,` *Position*`)` – Mark the digit at index *Position* in *Time* on *Date* with *Color*.
- `changecolor_day(`*Color*`,` *Date*`)` – Set the background color of *Date* to *Color*.
- `changecolor_week(`*Color*`,` *Date*`)` – Set the background color of all the days of the week in which *Date* is contained to *Color*.
- `changecolor_month(`*Color*`,` *Month*`)` – Set the background color of all the days of *Month* to *Color*.
- `changecolor_monthname(`*Color*`,` *Month*`)` – Mark the name (title) of *Month* with *Color*.
- `flash_text(`*Color*`,` *Date*`,` *TextString*`)` – Make *TextString* on *Date* blink in *Color*.
- `flash_text(`*Color*`,` *Date*`,` *Time*`,` *TextString*`)` – Make *TextString* at *Time* on *Date* blink in *Color*.
- `flash_time(`*Color*`,` *Date*`,` *Time*`)` – Make the *Time* label on *Date* blink in *Color*.
- `flash_digit(`*Color*`,` *Date*`,` *Time*`,` *Position*`)` – Make the digit at index *Position* in *Time* on *Date* blink in *Color*.
- `flash_day(`*Color*`,` *Date*`)` – Make the background color of *Date* blink in *Color*.
- `flash_week(`*Color*`,` *Date*`)` – Set the background color of all the days of the week in which *Date* is contained to *Color*.
- `flash_month(`*Color*`,` *Date*`)` – Make the background color of all the days of *Month* blink in *Color*.
- `flash_monthname(`*Color*`,` *Date*`)` – Make the name (title) of *Month* blink in *Color*.

4.21 InputOutputText Trindikit Module Agent

4.21.1 Functionality

The InputOutputText agent is a Trindikit agent written in java, which wraps two Trindikit modules, an input module and an output module. The user uses an input field for typing text input to the system. User and system utterances show up in a text log window.

The InputOutputText agent can be started in active mode and in passive mode. If started in active mode, it will continuously listen for input, if started in passive mode it will return the accumulated input so far when the `tkit_call_module(input,input)` solvable is called. The agent is designed to be compatible with the Nuance ASR and TTS module agents.

4.21.2 Links

http://www.ling.gu.se/projekt/talk/software/wrappers.html

4.21.3 Solvables

- `tkit_call_module(input,init)` – Only present for Nuance ASR agent compability reasons.
- `tkit_call_module(input,input)` (passive mode only) – Writes the text typed in the text input field to the Trindikit TIS and in the text log window.
- `tkit_call_module(input,quit)` – Only present for Nuance ASR agent compability reasons.
- `tkit_call_module(output,init)` – Only present for TTS agent compability reasons.
- `tkit_call_module(output,output)` – Reads output from the Trindikit TIS and writes it in the text log window.
- `tkit_call_module(output,abort)` – Only present for TTS agent compability reasons.
- `tkit_call_module(output,quit)` – Only present for backwards compability.

4.22 Nuance ASR Trindikit Module Agent

The Nuance ASR agent is a Trindikit agent written in java, which serves as a ewrapper to a Trindikit input module.
The Nuance ASR agent can be started in active mode and in passive mode. If started in active mode, it will continuously listen for speech input and write to the Trindikit TIS when speech is detected. If started in passive mode it will listen for speech input when the `tkit_call_module(input,input)` solvable is called and write the recognition result to the TIS.

4.22.1 Links

http://www.ling.gu.se/projekt/talk/software/wrappers.html

4.22.2 Solvables

- `tkit_call_module(input,init)` – Queries TIS for current language and current domain and loads corresponding speech recognition package.
- `tkit_call_module(input,input)` (passive mode only) – Calls recognition function and writes recognition result to the Trindikit TIS.
- `tkit_call_module(input,quit)` – deallocates resources.

4.23 TTS Trindikit Module Agent

The TTS module agent is a Trindikit agent written in java, which serves as a wrapper to a Trindikit output module. At startup a set of `language=port`-mappings are specified, so that several TTS servers for different languages can run simultaneously on different ports. Nuance Vocalizer is used as a text-to-speech synthesis server.

4.23.1 Links

http://www.ling.gu.se/projekt/talk/software/wrappers.html

4.23.2 Solvables

- `tkit_call_module(output,init)` – Queries TIS for current language and saves it as a local variable.
- `tkit_call_module(output,output)` – Reads output from the Trindikit TIS and synthesizes it using the TTS server for the current language.
- `tkit_call_module(output,abort)` – Stops TTS synthesis (barge-in).
- `tkit_call_module(output,quit)` – Only present for backwards compability.

4.24 DynGui Trindikit Module Agent

4.24.1 Functionality

The DynGui agent is a Trindikit OAA agent written in java, which acts as a dynamic GUI, displaying graphical output of a multimodal dialogue system. The DynGui reads output from the Trindikit TIS and parses it into GUI components, such as buttons, text fields etcetera. When the user clicks a button or types in a text field, input is written to the Trindikit TIS. The DynGui agent thus acts as a combined graphical input- and output module, communicating with Trindikit using the Trindikit OAA API described in section 3.3. The DynGui also allows the user to stop and start Trindikit and to change language.
For a longer description of the DynGui, see deliverable D1.6 ([5]).

4.24.2 Links

http://www.ling.gu.se/projekt/talk/software/wrappers.html

4.24.3 Solvables

- `tkit_call_module(output_gui,output_gui)` – Reads a description of the dynamically generated components from TIS and displays them.

4.25 FreeDB

The FreeDB Wizard is a multi-purpose tool for searching and manipulating a database of music information. It has two basic running modes for two separate purposes within TALK. First, it provides a GUI interface to the database which supports numerous types of searches on album information as well as building, searching, and manipulating playlists. This mode is used primarily in Wizard of Oz experiments to allow a human wizard to mimic system behavior by directly using the database. This mode can be configured to send messages over OAA when an operation on the database has occurred, so that other components can participate in the WoZ experiment.

The second mode is used within the dialogue system itself. It provides an SQL interface to the database over OAA. This can be used by the dialogue system to make a wide variety of queries to the database. The tool also allows playlist manipulation (e.g., new playlist creation, adding/removing songs, etc.) over OAA as well.

4.25.1 User Definable Knowledge Sources

FreeDB Wizard is patterned after the FreeDB music database `http://www.freedb.org`, which is a large open-source database of CDs and information about them (but not actual songs). The database includes information about genre, artist, album title, year, length, and track titles and lengths. The original database held over 600,000 albums, but as this information is contributed from a variety of sources (e.g., individual CD users), it has a number of problems, including duplicates. We have done a variety of automatic cleanup on the database and it now contains around 200,000 albums.

The FreeDB Wizard can import any user-defined text file in the form of FreeDB databases. It also supports exporting playlists or search results to a separate file, so that developers can easily construct subsets of the large database to use for their application.

4.25.2 Solvables

We only describe the OAA interface, as the GUI interface is fairly self-explanatory.

FreeDB accepts a single synchronous OAA solvable `freedb` which takes an SQL query string and returns a list of results. Each result is also a list of either a number (for `COUNT`) or column entries from the database. It is suggested that the number of results first be queried using `COUNT` and then only requested when it is certain that there aren't a very large number, which could slow down the system.

4.25.3 Use and Licenses

FreeDB Wizard was developed by CLT Sprachtechnologie, a subcontractor for USAAR as part of the TALK project.

CLT has licensed the FreeDB Wizard binaries to all TALK partners. It can be used not only by individuals in the project, but also anyone at the partner institution. This license is valid even after the TALK project ends. FreeDB Wizard may not be redistributed.

4.26 Mary Speech Synthesis Wrapper

This agent encapsulates the Mary speech synthesis system (developed by DFKI) by connecting a Mary client to OAA. It expects input in MaryXML markup and transfers it to a Mary server. It then handles the local sound output for the response from the Mary server. Markup in MaryXML allows the control of the output language, including arbitrary switching between German and English in the same utterance.

We added user defined dictionaries for correct pronounciations of one language (e.g., English song, album and artist names) in output sentences of the other language (e.g., German). The Mary Agent was also extended to automatically reduce the volume of the Mp3 player while synthesized speech is output to the audio channel.

Links

- Mary speech synthesis system: `http://mary.dfki.de/`

4.27 Nuance `nl-tool` Wrapper Agent

The command-line utility `nl-tool` from Nuance is shipped with Nuance version 8.5.0. It takes a line of input text (via standard input) and outputs the corresponding interpretation(s) returned by running a given Nuance grammar on that text input (exactly in the same way it would be run if that text input had been recognized by the Nuance speech recognizer).

The Nuance `nl-tool` Wrapper Agent wraps this utility and allows it to be called from within an OAA community. This can then be used to support text input to a dialogue system which uses Nuance grammars within the recognizer to do the first pass of natural language understanding (using Nuances NLU support).

4.27.1 User Definable Knowledge Sources

Nuance `nl-tool` takes a compiled Nuance package (grammar) as input.

4.27.2 Solvables

The `nl-tool` wrapper uses the following solvables:

- `nuance_analyze(Text,Result)`: a synchronous call which sends a text string to the recognizer and returns an icl-ified version of the parse/NLU result.
- `nuance_nl_change_package(Package, Grammar)`: used to switch the package and/or grammar to be used by `nl-tool` on the next invocation of `nuance_analyze`.

4.27.3 Use and Licenses

The OAA wrapper for Nuance nl-tool was developed for TALK. It is available under an open-source license.

4.28 Nuance Monitor Agent

The Nuance monitor agent is a special speech recognition control and handling tool for the SAMMIE systems implemented as an OAA agent. It manages the interface between the Nuance wrapper agent and the SAMMIE system and is responsible to realize the automatic closing and opening of the microphone. The Nuance monitor operates in an endless loop that monitors Nuance events. This loop simply consists of starting the recognition process, asking for the recognition result and processing the result. Starting the recognition opens the microphone. The Nuance wrapper returns with either a recognition result or a no speech timeout. In the latter case the microphone is simply closed by not entering the loop again and this is forwarded to the GUI agent in order to switch the color of the microphone button correspondingly. If there are recognition results the microphone is kept closed as well, and the recognition results are forwarded to the interpretation. As soon as the system forwards readiness for further interaction, the mentioned endless loop is entered which opens the microphone again.

4.29 SVOX Synthesis Wrapper

The freely available speech synthesis systems have proven to be potentially distracting for the driver due to their bad synthesis quality. As we intended to evaluate the SAMMIE in-car system in naturalistic driving tests, BMW has provided a commercial synthesis system from SVOX AG (http://www.svox.com/) for the SAMMIE system integrated into the BMW car. The SVOX TTS has been specifically developed for BMW outside of TALK and is not part of the TALK project.

We implemented and used a wrapper for SVOX, which was written in Java and C++ using JNI and serves the same OAA interface as the Mary synthesis. The agent is capable to handle corresponding solvables for (i) synthesising, (ii) language switching (German and English), (iii) system start, and (iv) immediate output stop.

4.30 Loquendo Synthesis Wrapper

In order to realize better speech synthesis output quality we alternatively implemented and used a multilingual Loquendo TTS wrapper. The wrapper was written in C++ and serves the same OAA interface as the Mary synthesis. The agent is capable to handle corresponding solvables for (i) synthesising, (ii) language switching, (iii) system start, and (iv) immediate output stop.

4.31 Loquendo ASR Wrapper

For the SAMMIE MP3 system we also implemented an alternative ASR wrapper agent using the Loquendo ASR. The agent was implemented in C++ and provides the same OAA interface as the Nuance agents. The only difference to Nuance is that the Loquendo wrapper supports the W3C standards compliant SRGS (speech recognition grammar specification) grammar format.

4.32 iCOMM Wrapper

The IComm (stands for **I**nternal **Comm**unication) agent implements the OAA interface to the following PATE-based core modules of the SAMMIE system (see D5.3 for details):

Interpretation Manager

Dialogue Manager

Discourse and Context Manager

Presentation Planner

All listed components run in the same virtual machine, whereas each of them runs in separate thread. The IComm agent administrates the communication between the modules themselves and the provides an interface to agents within the oaa community. Communication between the IComm agent and other oaa agents is usally *string*-based. The internal communication between the wrapped modules is *object*-based.

4.32.1 Solvables

- *makeInterpretation(XMLMessage)*: sends the ASR output to the interpretation manager.
- *handleGraphicalInput(XMLMessage)*: sends the model of a selected graphical item to the interpretation manager.
- *switchDebugGUIs(Mode)*: changes the debug state of the wrapped system components. *Mode* can be set to *true* or *false*. In addtion each of the modules wrapped by IComm can individually be switched by the separate solvables.
- *queryInformationState(XMLMessage)*: allows other modules within the oaa community to query the information state represented by the Discourse and Context Manager.
- *storeModuleInformation(XMLMessage)*: allows other modules within the oaa community to store information in the information state.
- *language-switch(Language)*: informs IComm about a change in of the language type. Possible values are englisch ('en') or german ('de').
- *system-reset*: resets the state of the modules wrapped by the IComm agent.

4.33 The Dialogue Control Agent

The dialogue control agent (DCA) is a special system control tool of the SAMMIE system that serves multiple purposes at once. It allows for immediate system access, control and administration from outside the multimodal human machine dialogue.

The following figure 4.2 shows the GUI of the DCA of the baseline system. It was quite simple as it just provided general control buttons to (i) reset the system, (ii) pause the system (i.e., close the microphone), and (iii) start a new dialogue which triggered the initial greeting message. Technically speaking, all

DCA buttons call corresponding OAA solvables which, in turn, must be appropriately processed by the respective (i.e., affected) modules. For example, the 'reset' event must be handled appropriately by every module as it requires the return to the initial state while the 'pause' event only affects the modules that manage speech recording tasks.

Figure 4.2: GUI of the Dialogue Control Agent for the Sammie MP3 Baseline System

For the evaluation system of the SAMMIE final showcase, we implemented a highly enriched DCA as the GUIs of the evaluation system in figure 4.3 and of the multilingual system in figure 4.4 show. There is now a simplified system architecture showing buttons for the major components which, in turn, dynamically switch color while they are active to provide an 'online' visualisation of the distribution of processing within the SAMMIE system. Furthermore, the DCA dynamically shows the last speech input and automatically measures the response times of graphical and speech output. Finally, there is now a series of control buttons available serving various purposes:

- System reset.
- Pause/resume speech recording (manually close/open the microphone, same functioning as the steering wheel buttons in the car).
- Trace mode switching on/off.
- Language switching between German and English.
- System version switching (full, non adaptive, command & control).
- Evaluation logging switching on/off.
- Debugging GUI switching on/off.
- Starting a new dialogue.
- Immediate stoppping of a running synthesis.
- Manual fine tuning of speech recognition parameters (end of speech silence, no speech timeout, threshold of the signal-to-noise ratio).
- Triggering the summarisation message (summaries not implemented in the system).

The DCA GUI of the German evaluation system shown in figure 4.3 omits the buttons for language switching.

For the multilingual SAMMIE MP3 system we used a slightly modified DCA version that just differs in the available control buttons. As shown in figure 4.4, this DCA GUI omits the system version switching buttons whereas it contains the buttons for language switching.

To sum up, the DCA was a very helpful tool that could be used for multiple purposes at once:

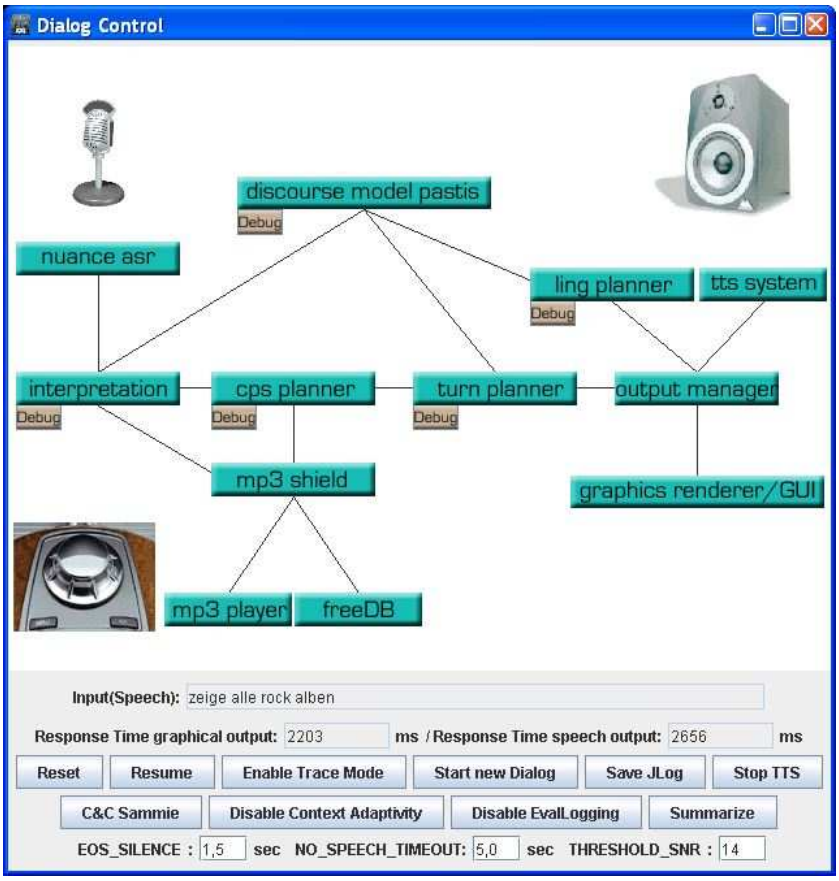


Figure 4.3: GUI for the Dialogue Control Agent for the German Sammie MP3 Final Showcase Evaluation System

- Valuable and comfortable tool to control and manage the system during runtime “from outside” in implementation, integration, and testing phases.
- Visualisation and system control during demonstrations.
- Comfortable system access and control for the experiment conductor during evaluation experiments.

4.34 RapidFire

RapidFire consists of two subparts which support simple mappings for interpretation and generation in dialogue for rapid prototyping of dialogue systems, as well as concurrent development of the dialogue system and the interpretation and generation components.

RapidFire$_i$ is the interpretation module. It supports user-defined mapping rules from strings to dialogue acts. A rule has two parts: the first defines a string template, which is a regular expression-like string

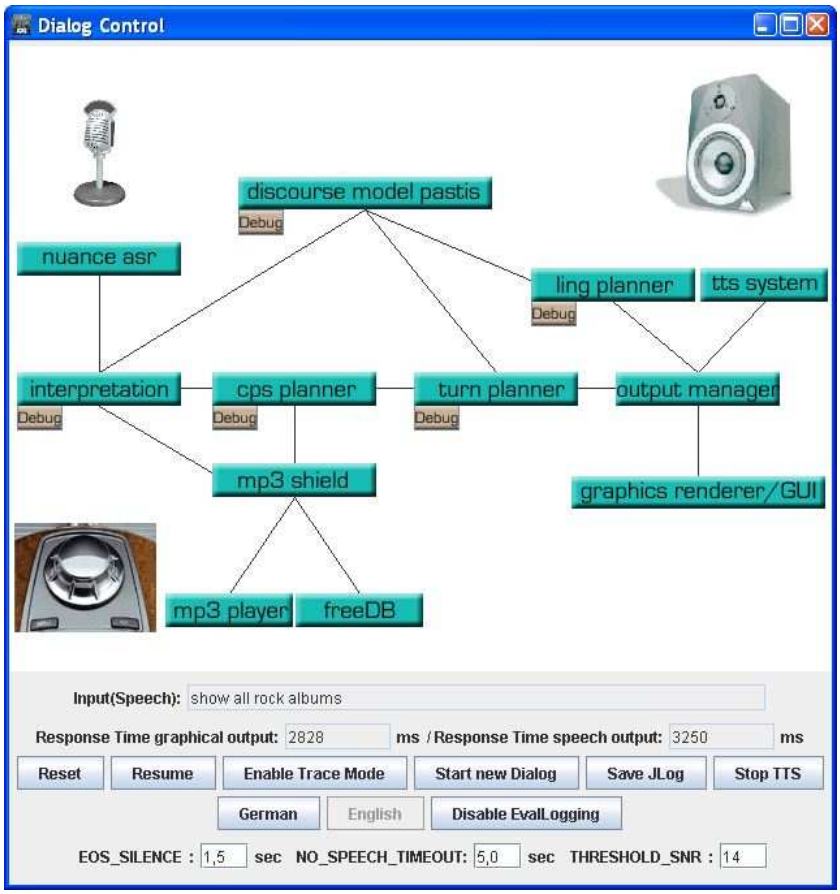


Figure 4.4: GUI of the Dialogue Control Agent for the Multilingual Sammie MP3 System

which supports both variables and optional parts. The tail of the rule is a list of dialogue act templates which this string should be mapped to. These also can use variables defined in the string template so that variables will be passed to the dialogue act generated itself.

When a new string is received, it is matched against the string template of each rule. If the string matches, variables are instantiated given the actual string, and then the list of dialogue acts is generated and instantiated with variables from the string template. RapidFire$_i$ then outputs a list of lists of dialogue acts for each rule that matched the input.

RapidFire$_g$ is the generation module. It works more or less in the reverse way from RapidFire$_i$. It receives a dialogue act and then uses rules to map it into a string. Again templates are used with variables so that the string can be instantiated with the variable value from the dialogue act.

This provides a simple way to provide some coverage for both interpretation and generation to rapidly prototype dialogue systems, and allows one to produce a full working system early on in the development process, even before full-fledged interpretation and generation components are functional, thus supporting parallel development.

4.34.1 User Definable Knowledge Sources

Mapping rules are defined in separate files for both RapidFire$_i$ and RapidFire$_g$.

4.34.2 Solvables

The RapidFire modules can either be used through an OAA interface. Over OAA, RapidFire$_i$ receives a string and returns a list of lists of dialogue acts. RapidFire$_g$ receives a dialogue act and returns a string.

RapidFire$_i$ takes a synchronous OAA solvable `rapidfire_i` containing a string and it returns a list of lists of instantiated dialogue acts.

RapidFire$_g$ takes a synchronous OAA solvable `rapidfire_g` with a dialogue act and returns a string.

4.34.3 Use and Licenses

RapidFire was developed as part of TALK and is available under an open-source license.

Chapter 5

Systems

A series of complete dialog systems was developed and implemented in the TALK project. In this section we describe selected versions of systems, each of which using some of the infrastructure libraries described above in this document. For each of the systems below there is also a classification that shows which of the components described in the above sections were used, respectively, and how.

These systems implement different research issues in it at once. Details on these systems including the respective resarch issues addressed in each system can be found in the respective deliverables. Shorter descriptions on earlier versions of these systems are also available in status report T5.3s2.

5.1 UEDIN-UCAM In-car information System

The dialogue system of [11, 10] was built around the DIPPER dialogue manager [1]. We modified the DIPPER dialogue manager so that it can consult learnt strategies (for example strategies learnt from the 2000 and 2001 COMMUNICATOR data [9]), based on its current information state, and then execute dialogue actions from those strategies. This allows us to compare hand-coded against learnt strategies within the same system (i.e. the other components such as the speech-synthesizer, recogniser, GUI, etc. all remain fixed).

5.1.1 A system exhibiting Reinforcement Learning

The central motivation for building this dialogue system was as a platform for Reinforcement Learning (RL) experiments. The system exhibits RL in 2 ways:

- It can be run in online learning mode with real users. Here the RL agent is able to learn from successful and unsuccessful dialogues with real users. Learning will be much slower than with simulated users, but can start from an already learnt policy, and slowly improve upon that.
- It can be run using an already learnt policy (e.g. the one reported in D4.1 [9], learnt from COMMUNICATOR data). This mode can be used to test the learnt policies in interactions with real users.

Please see TALK deliverable D4.1 [9] for an explanation of the techniques developed for Reinforcement Learning with ISU dialogue systems.

5.1.2 Overview of system features

The following features were implemented in the UEDIN/UCAM system:

- Interfaced to learnt dialogue policies
- Multiple tasks: information seeking for hotels, bars, and restaurants
- Basic slot-filling dialogues
- Overanswering/ user-initiative
- Open speech recognition using n-grams
- Use of dialogue plans
- Open-initiative initial system question
- Basic user-goal/task recognition
- Confirmations - explicit and implicit based on ASR confidence
- Fragmentary clarifications based on word confidence scores
- Template-based descriptions of database entities
- Multimodal output - highlighting and naming entities on GUI
- Startover, quit, and help commands
- Simple user commands (e.g. “Show me all the indian restaurants”)
- Logging in TALK ISU format

5.1.3 System components

The major components are:

- DIPPER [1] for ISU-based dialogue management[1]
- ATK for speech recognition[2]
- Dialogue Policy Learner Agent[3]

[1] Both Java and Prolog version are available, with OAA wrappers.
[2] See `http://mi.eng.cam.ac.uk/~sjy/software.htm` . OAA wrapper developed in C++.
[3] This is written in Python and has an OAA wrapper in C.

- Festival2 [15][4]
- Display agent (an OAA agent written in java)
- Database agent (an OAA wrapper to MySQL, written in java)

These components are described in detail in D4.2 [11].

5.1.4 Summary

The main achievements made in designing and constructing this baseline system were:

- Combining learned dialogue policies with an ISU dialogue manager. This has been done for online learning, as well as for strategies learned offline.
- Mapping learned policies between domains. i.e. mapping Information States and system actions between DARPA COMMUNICATOR and in-car information seeking tasks.
- Fragmentary clarification strategies: the combination of ATK word confidence scoring with DIPPER ISU-based dialogue management rules allows us to explore novel clarification techniques.
- Software integration: building a functioning multimodal dialogue system with the ATK, DIPPER, MySQL, Festival, java DisplayAgent, and learned dialogue strategy components.

5.2 The USE MIMUS System

As mentioned in previous reports, 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, USE has focused on wheel–chair bound users and their special circumstances.

In this particular scenario, users are able to access the system at all times through different modalities, that is, using speech and/or a graphical interface. The scenario includes microphones, speakers and a touch screen where the information can be displayed and introduced or selected.

The software components of the USE system are implemented as independent OAA agents, linked therefore through the OAA facilitator. An overall view of the system is shown in figure 5.1.

The core of the System is the Dialogue Manager agent whose role is to control the course of the interactions with the user, checking the Multimodal Input Pool for new inputs from the user, that may come from the ASR agent (speech, currently the OAA wrapper for Nuance) or from the Home Setup agent (clicks). The presentation of information from the Dialogue Manager to the user can also be done multimodally: by voice through the Talking Head (using Loquendo's TTS) and graphically through the Home Setup agent as well as the display agents. The Talking Head acts also as a channel for graphical information to the user, expressing surprise, happiness, etc.

The Dialogue Manager uses the Knowledge Manager to perform reference resolution to locate specific devices. The latter may be referred to by its label, as well as by its location, type, etc.

Finally the Dialogue Manager may decide to execute commands (e.g. switch on a light) using the x10 protocol through the Device Manager.

[4] OAA wrapper developed in C.

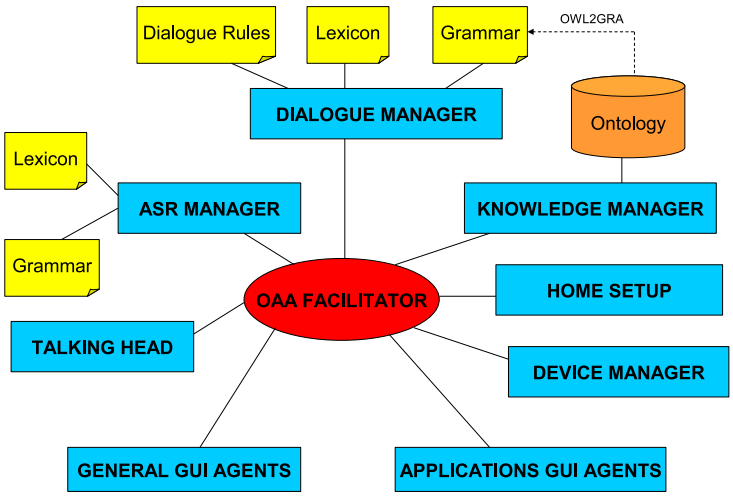


Figure 5.1: Architecture

5.3 DFKI/USAAR SAMMIE Wizard-of-Oz and Dialog Systems

The SAMMIE Wizard-of-Oz system is described in detail in the deliverables D6.2 and D6.4 as well as in [7]. The software components used in that setting are:

- FreeDB agent
- Table presenter
- Options presenter
- Keyboard input agent
- Synthesis (Mary)

The in-car baseline showcase system is described in deliverable D5.2 and in chapters 1 and 2 of the second year status report T5.3s2 for task 5.2. The following list shows the software components used in that system:

- Nuance Wrapper
- Nuance Monitor
- DIPPER dialog manager
- Turn planner

- Output manager
- MP3 shield
- FreeDB agent
- jlGui Wrapper agent
- Multifunctional GUI with player GUI and table presenter
- Pastisse discourse manager
- Interpretation manager
- Generation manager
- Linguistic planner
- Synthesis agent (Mary)

The in-car final showcase system is described in detail in Deliverable D5.3. The software components used in that system are:

- Nuance Wrapper
- Nuance Monitor
- IComm dialog manager
- Output manager
- MP3 shield
- FreeDB
- jlGui player
- Multifunctional GUI
- Linguistic planner
- Synthesis (Mary, Loquendo, SVOX)

5.4 The UGOT Systems

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, which does not use the GoDiS dialogue manager, described below.

The application uses the GoDiS dialogue manager and the Trindikit4 dialogue system toolkit and consists of a collective of OAA agents The Controller agent, DME agent and the MMD, ASR and TTS agents are Trindikit agents, which communicate using the TRINDIKIT 4 OAA API.

- 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 users'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 continuosly 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.

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.
The application uses the GoDiS dialogue manager and the Trindikit4 dialogue system toolkit and consists of a collective of OAA agents. The Controller agent, DME agent and the MMD, ASR and TTS agents are Trindikit agents, which communicate using the Trindikit4 OAA API.

- 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 users'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 continuosly 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 DeLux Gui agent displays a schematic map of the house showing each lamp in its specified locaiton. According to the user actions the Gui is modified to reflect the current status of each lamp in the house.

DJ-GoDiS

The DJ-GoDiS application is a multimodal interface to a MP3 player, built using GoDiS and TrindiKit4, a new distributed asynchronous version of TrindiKit, using OAA to communicate between system components.

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.

DJ-GoDiS uses integrated multimodal input and parallel multimodal output. The asynchronous control of TrindiKit4 enables the GUI and the ASR components to operate independently from each other, while still writing their input to the same Information State variable, a queue. The contents of the input queue are copied to an input variable at the end of the user turn. System output is presented in parallel as speech and as graphical GUI components of the GUI.

The DJ-GoDiS consists of a number of OAA agents the execution of which are controlled by a single agent, the controller. The current agent setup is:

- OAA facilitator – routes OAA calls to an from other agents
- Control Agent – executes a Trindikit control algorithm
- DME agent – contains the GoDiS dialogue move engine and the interpretation and generation modules
- JLGuiAgent – a OAA wrapper to a mp3 player
- PlayerGUI – the DJ-GoDiS GUI which serves as a graphical input and output module
- Alarm agent – used for determining when the user's turn has been ended
- InputNuance – ASR module using Nuance for speech recognition
- TTS Agent – TTS module using Nuance Vocalizer for text-to-speech synthesis

A more thourough description of DJ-GoDiS is given in deliverable D1.6.

AgendaTalk

The UGOT calendar application, AgendaTALK, has been built with the GoDiS dialogue system and TrindiKit toolkit and works as a voice interface to a graphical calendar application in the in-home environment but could also be used in an in-car environment.

The user can manage his schedule using the voice by adding, deleting, changing, or searching information in the calendar. Alternatively, the user can access the graphical interface directly like in a standard desktop calendar application. Both modes work in parallell and are connected by using the same database of calendar information. However, integrated multi-modality is not possible in this application.

Communication between components is handled by OAA's asynchronous hub architecture, and all components currently run under Windows and is started up with the OAA Start-It agent. The major components are:

- Nuance for speech recognition.
- GoDiS for ISU-based dialogue management
- TrindiKit SQL resource for interaction with calendar database
- BORG Calendar Application as Calendar GUI (see http://borg-calendar.sourceforge.net/)
- Nuance for speech synthesis.

The graphical calendar system chosen to use with AgendaTALK is the calendar system BORG (http://borg-calendar.sourceforge.net/) which is an open-source calendar application written in Java. AgendaTALK shares calendar database with the BORG system by altering the same MySQL database and in that way they share the same information. The AgendaTalk application interacts with the database via the TrindiKit SQL resource. An OAA wrapper processing iCal files, the iCal Agent, has also been developed to not close doors for other calendar applications to be connected with AgendaTalk.

AgendaTALK and its components are described further in deliverable T1.6s2. The current state is that we have an end-to-end system working in both English and Swedish connected with a graphical interface through a calendar database. The functionality supported in the current version is:

- Add restricted types of events such as meetings, appointments, presentations etc. to the calendar
- Change time or date for an event in the calendar
- Delete an event from the calendar
- Ask for the time of a certain event
- Ask if booked a certain time or date
- Ask for all bookings a certain day
- Ask for today's date

Further work will be to improve the application during the following year and add the functionalities missing. We will also spend some time on improving the speech recognition and the interpretation for the domain by carrying out the experiments described in D1.3 using GF also for this domain.

UGOT Tram Information System (GOTTIS)

The Göteborg Tram Information System (GOTTIS) is a demonstration of a multilingual multimodal dialog system. It finds the shortest path through (a subset of) the Göteborg public transportation network. User input consists of spoken queries along with clicks on a map of the transport network. The system responds with spoken instructions and drawings on the network map. The system can be used in Swedish or English.

The system is easily adaptable to other transport networks or other systems which can be represented as weighted directed graphs.

The application consists of a number of OAA [12] agents:

- Speech recognizer - The Nuance [13] speech recognizer using NuanceWrapper [6]. The speech recognition grammar is generated from the GF user grammar.
- Clickable map and path drawing - MapAgent [3], an OAA agent written in Java.
- Parser and linearizer (multilingual and multimodal) - The Embedded GF Interpreter through its OAA interface.
- Shortest path finder - OAA agent written in Java.
- Speech synthesis - FreeTTS [14] over OAA using FreeTTSAgent [4] (English), or Nuance over OAA using NuanceWrapper (English and Swedish).

The system is described in more detail in deliverable D1.2a.

Chapter 6

Conclusion

As a service package, the main objective of WP5 was providing a common software basis for libraries that are developed in WPs 1-4 and their integration into showcase and laboratory systems. This deliverable describes how task 5.1, “Infrastructure”, has successfully provided this software basis. The integration into showcase systems and other laboratory systems is described in deliverables D5.2 and D5.3 and further deliverables in workpackages 1-4.

The TALK project software infrastructure has been developed in three steps. First, a common middleware has been selected as the framework for implementing modularized multimodal dialogue systems. As described in deliverable D5.1.1, we have collected a set of requirements from all partners concerned and finally chosen the Open Agent Architecture (OAA) as the common framework throughout the TALK project. This has enabled flexible design and development of systems, allowing modules to be exchanged and added easily. It has transparently allowed us to run systems in a distributed fashion on multiple machines and also using a well-established middleware like OAA has provided a number of development and debugging tools. However, we have still added a number of new tools and extensions to OAA, e.g., advanced logging mechanisms. The resulting middleware and module communication infrastructure has provided to necessary flexibility and efficiency for our real-time showcase systems.

Second, we have made a sizeable number of existing and new modules middleware-compliant, usually by adding an ‘OAA wrapper’. This has allowed the reuse or exchange of modules in various OAA-based systems. Many of these modules are built on existing applications like speech recognisers and speech synthesisers. However, these wrappers are a significant contribution by TALK to the community at large since they make these modules available to other developers in the OAA community.

Third, we have designed and implemented a number of generic modules that address common tasks that occur in more than one of our multimodal dialogue systems, such as the use of databases and various graphical presentation modules.

Finally, all of our modules–supported by the complex message types of OAA–communicate with well-defined interfaces and declarative representations of data, typically based on XML-languages. Where possible, we have used common standards across modules, such as Typed Feature Structures and OWL.

Bibliography

[1] Johan Bos, Ewan Klein, Oliver Lemon, and Tetsushi Oka. Dipper: Description and formalisation of an information-state update dialogue system architecture. In *4th SIGdial Workshop on Discourse and Dialogue*, Sapporo, Japan, 2003.

[2] Johan Bos and Tetsushi Oka. Building spoken dialogue systems for believable characters. In *Proceedings of the seventh workshop on the semantics and pragmatics of dialogue (DIABRUCK)*, 2003.

[3] Björn Bringert. MapAgent, December 2004. `http://www.cs.chalmers.se/~bringert/gf/map-agent.html`.

[4] Håkan Burden. *FreeTTS Agent for OAA*. Göteborg University, Gothenburg, Sweden. `http://www.ling.gu.se/projekt/talk/software/reports/freeTTSAgentReport.pdf`.

[5] Stina Ericsson, Gabriel Amores, Björn Bringert, Håkan Burden, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonsson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, Guillermo Pérez, and Mikael Sandin. Software illustrating a unified approach to multimodality and multilinguality in the in-home domain. Deliverable D1.6, TALK Project, 2006.

[6] David Hjelm. *NuanceWrapper manual*. Göteborg University, Gothenburg, Sweden, June 2004.

[7] Ivana Kruijff-Korbayová, Nate Blaylock, Ciprian Gerstenberger, Verena Rieser, Tilman Becker, Michael Kaißer, Peter Poller, and Jan Schehl. An experiment setup for collecting data for adaptive output planning in a multimodal dialogue system. In Graham Wilcock, Kristiina Jokinen, Chris Mellish, and Ehud Reiter, editors, *Proceedings of the 10th European Workshop on Natural Language Generation (ENLG-05)*, pages 191–196, Aberdeen, Scotland, August 8–10 2005.

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

[9] Oliver Lemon, Kallirroi Georgila, James Henderson, Malte Gabsdil, Ivan Meza-Ruiz, and Steve Young. D4.1: Integration of Learning and Adaptivity with the ISU approach. Technical report, TALK Project, 2005.

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

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

[12] David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1–2):91–128, January–March 1999.

[13] Nuance Communications, Inc., Menlo Park, CA, USA. *Nuance Speech Recognition System 8.5: Introduction to the Nuance System*, December 2003.

[14] Sun Microsystems, Inc. *FreeTTS Programmer's Guide*, 2003.

[15] P. Taylor, A. Black, and R. Caley. The architecture of the the Festival speech synthesis system. In *Third International Workshop on Speech Synthesis, Sydney, Australia*, 1998.

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

Appendix: Software on DVD

The software described in this deliverable and developed in the TALK project is attached here as a DVD-R. See the `README` file for more information.