D5.1.1: Infrastructure

Tilman Becker (ed.), Peter Poller, Staffan Larsson,
Björn Bringert, Håkan Burden, Carine Cassia,
Ann-Charlotte Forslund, David Hjelm, Rebecca Jonsson,
Andreas Wallentin, Anna Wählby, Oliver Lemon,
Kallirroi Georgila, Guillermo Pérez,
Nate Blaylock, David Milward

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable 5.1.1

February 14th, 2005

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	M12 = December 2004
Actual date of delivery	February 14th, 2005
Deliverable number	5.1.1
Deliverable title	D5.1.1: Infrastructure
Type	Report
Status & version	Final 1.0
Number of pages	44 (excluding front matter)
Contributing WP	5
WP/Task responsible	DFKI
Other contributors	
Author(s)	Tilman Becker (ed.), Peter Poller, Staffan Larsson, Bj¨orn Bringert, Håkan Burden, Carine Cassia, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonsson, Andreas Wallentin, Anna W¨ahlby, Oliver Lemon, Kallirroi Georgila, Guillermo Pérez, Nate Blaylock, David Milward
EC Project Officer	Kimmo Rossi
Keywords	software infrastructure, middleware, software agents

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 Saarbr¨ucken, 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

Executive summary

This report details work done in the first year of the TALK project to establish an appropriate software infrastructure for multimodal dialogue systems and their components, with respect to the TALK project research goals.

Section 1 gives an introduction to the specific tasks within Task 5.1 in the first year.

Section 2 discusses the need for a common middleware architecture and the choice process in the TALK project that has led us to use the Open Agent Architecture (OAA).

Section 3 describes our progress in connecting existing and new modules to the OAA middleware.

Section 4 presents our work on general purpose modules that are intended to be used in multiple systems. They too are communicating as OAA agents.

Finally, section 5 briefly notes on two examples of the systems that are built on the TALK architecture as described in the previous sections. More information on systems can be found in T5.2s1 and the reports on workpackages 1, 2, 3, and 4.

Following the conclusion and a sketch of our timeline for the next year in section 6, an appendix lists the questionnaire we used to collect an overview over existing software and their infrastructure and integration requirements.

Chapter 1

Introduction

The TALK project is concerned with the design of multimodal dialogue systems where speech is a primary input modality, with gestures (mouse, joystick, pen) as and additional input modality, and a mixture of speech and graphics is used for dialogue system output. The project encompasses research and development of various aspects of such a system as well as the development of baseline and laboratory systems to perform evaluations on an end–to–end system level.

This report details work done in the early phases of the project to establish an appropriate software infrastructure to support the development of such complete systems. The block diagram of a typical multimodal dialogue system is shown in Figure 1.1. As can be seen, such a system consists of a large number of components, each of which is already relatively complex. In any given system, a number of such modules will be implemented with already existing components. Therefore we need a flexible architecture to integrate new and existing modules, as well as identify and provide generic modules that are needed for multiple systems.

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

1. Find a suitable middleware connecting the components of a system.
2. Make existing (and new) modules accessible for this middleware.
3. Provide general purpose modules.

Finally, we report briefly on progress made in task 5.2, “Integration,” since only the successful integration of the components into complete systems validates our approach to modularization and to connecting the modules.

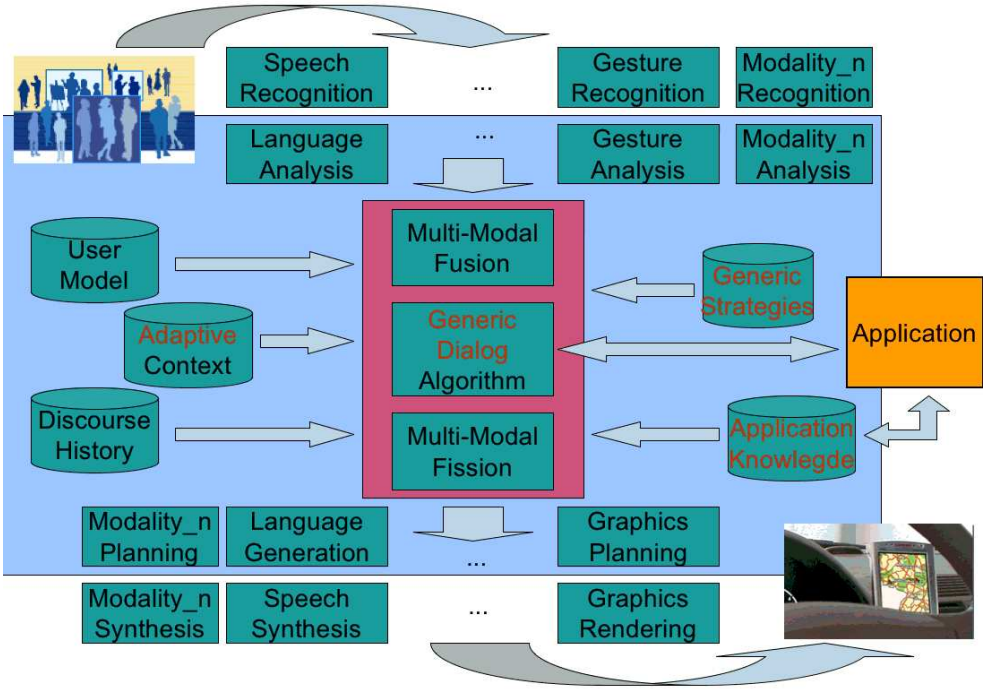


Figure 1.1: Main components in a multi-modal dialogue system.

Chapter 2

Middleware: The Open Agent Architecture OAA

From a requirements analysis for a middleware and a review of middleware architectures already in use at the partner's sites, we have identified the Open Agent Architecture (OAA), see [MCM99], as the common middleware that will be used in all systems that are developed within TALK.

2.1 Module Communication

As outlined in the introduction, systems in TALK will be distributed, multimodal, multi-component systems. Therefore we need a software architecture that enables and supports agile, flexible and dynamic composition of the modules and hides as many details of the communication protocol as possible from the module programmers.

This section gives a short overview of the major distributed communication frameworks that can be used to spread interacting components of a large system across multiple computers to achieve distributed processing. See also [Pol02] for more details.

First of all, there are large number of basic communication frameworks in which the interactions between components are preconfigured, i.e., hard-coded. All details of the communication protocols have to be maintained by the programmers themselves. An example of such a hard-coded communication model is Sun's Jini. It realizes only a simple infrastructure to establish a connection over a network.

There are more sophisticated and comfortable distributed models which include an additional communication layer, the so called 'middleware', which offers a more powerful mechanism to create distributed applications based on 'meaningful' communication objects and hiding some of the more complex protocol tasks from the programmers. Examples are the Common Object Request Broker Architecture (CORBA) or Microsoft's Distributed Component Object Model (DCOM). A disadvantage of these communication frameworks is that they still need fixed point-to-point communication in the sense that communication partners have to know each other.

Finally, there are communication architectures including a middleware layer in which a component can publish requests without any knowledge about the component that fulfills this request. In such architectures, data producers are decoupled from data consumers. There are two communication models that fulfill this requirement: the blackboard metaphor and the delegated computing model in which a central

message brokering unit is responsible for maintenance, coordination, and delivery of requests and their responses to the correct recipients.

2.2 Middleware: Blackboard and Distributed Computing

The *blackboard approach* is a widely used architecture, and consists of three parts: (i) a set of independent modules which provide the expertise needed to solve the problem; (ii) a blackboard, which is a shared global database which the modules send messages to and receive messages from; and (iii) a control component, which makes runtime decisions about the resource allocation and the order in which the modules operate on the blackboard. Initially, each module informs the blackboard about the kind of events it is interested in and contributes with its expertise to the solution of the overall problem which leads to an incremental solution of the problem. The entire communication between the modules takes place solely through the blackboard. This database encompasses the problem-solving state data, e.g., the hypotheses, the partial solutions, and, more generally, all data exchanged between the modules. The Multiplatform system [HNM+04] is an example for the blackboard approach.

In a framework based on the *delegated computing model* there is a specialized central brokering unit that coordinates the activities of the individual modules. At the beginning, each module informs the central broker about its capabilities. Service request messages can then be published to the broker whose task it is to maintain their processing, i.e., the delegation of the request to the recipient(s), the coordination of their efforts and the delivery of the result(s) to the requester. The Open Agent Architecture is an example for the delegated computing model.

Architectures that are based on these elaborated computing models do not only contain the necessary communication infrastructure. They are complete integration platforms that significantly support the complex process of integrating a new module into an overall system of communicating modules. Furthermore these architectures include several helpful tools, e.g., a graphical user interface (GUI), methods for easy debugging, and tools for monitoring and logging. They are sophisticated architectures that have been successfully used in several complex systems.

In the TALK project, such an easy-to-use architecture of either of these two kinds is needed because it hides time-consuming interface and communication programming from the module developers to a significant degree.

Selection criteria for a middleware are whether modules needed for a multimodal dialogue system can be included in the architecture, or are–ideally–already made available. A further criterion is the experience that partners already have with a particular architecture. Finally, the middleware architecture should be freely available either as open source or by one of the project partners.

2.3 Middleware Candidates

Given the requirements outlined in the previous section, the following middleware architectures are relevant for TALK:

- Galaxy Communicator
- Multiplatform

- Open Agent Architecture (OAA)

2.3.1 Galaxy Communicator

The Galaxy Communicator is an open source system architecture that enables developers to combine architecture-compliant commercial software and research components. It is a distributed, message-based, hub-and-spoke infrastructure that was especially optimized for spoken dialog systems. The Communicator infrastructure is an extension and evolution of the MIT Galaxy System, and is being developed, maintained, and provided by the MITRE Corporation. Version 4.0 contains APIs for C, C++, Java, Python and Allegro Common Lisp. It runs under Sparc Solaris, Intel Linux and Windows.

An instance of a Communicator infrastructure consists of arbitrary many processes that may be running on separate computers. The processes are arranged in a hub-and-spoke configuration, which means that a central processing unit (the hub) is used to mediate message based connections between communicating servers/modules (the spokes). Communicator provides the commonly used tools for control, logging and debugging.

In the TALK project, only some of the partners have immediate, existing work based on this architecture.

2.3.2 Multiplatform

Multiplatform is a multi-blackboard system architecture for distributed systems that consist of independent cooperating modules which may be running on separate machines under different operating systems [HNM+04]. Originally developed in the VerbMobil project, DFKI continuously improved and extended the Pool Architecture for the multimodal SmartKom dialogue system. There are module APIs for the following programming languages: C, C++, Java and PROLOG. The data delivery is based on PVM (Parallel Virtual Machine). The architecture itself runs under UNIX, LINUX and Windows.

Modules are realized as independent processes that interact by publishing and subscribing data to and from global data stores (so-called pools). This makes arbitrary communication paths in a dynamic community of distributed processes possible because data producers and data consumers are completely decoupled. The architecture does not only consist of the necessary infrastructure for inter-process communications. It also contains several kinds of helpful tools for integrators as well as for module developers.

In the TALK project, only DFKI has immediate existing work and extensive experience based on this architecture.

2.3.3 Open Agent Architecture

The Open Agent Architecture (OAA) is a framework for integrating a community of heterogeneous software agents in a distributed environment. The architecture is developed and maintained by SRI, Menlo Park, CA. The architecture runs under Unix, Linux and Windows. It contains APIs for C, (Visual) C++, PROLOG and Java.

In OAA, an agent is defined as any software process that meets the conventions of OAA. A key distinguishing feature of OAA is its delegated computing model that enables both human users and software agents to express their requests in terms of what is to be done without requiring a specification of who is to do the work or how it should be performed. Similar to the Galaxy Communicator there is also a

central processing unit, the so called facilitator, which is a specialized server agent within OAA that coordinates the activities of agents for the purpose of achieving higher-level, often complex problem-solving objectives. The knowledge necessary to meet these objectives is distributed in four locations in OAA:

- requester: It specifies a goal to the facilitator and provides advice on how it should be met.
- providers: They register their capabilities with the facilitator, know what services they can provide, and understand limits on their ability to do so.
- facilitator: It maintains a list of available provider agents and a set of general strategies for meeting goals.
- meta-agents: They contain domain- or goal-specific knowledge and strategies that are used as an aid by the facilitator.

Based on this knowledge, the facilitator matches a request to one or more agents providing that service, delegates the task to them, coordinates their efforts, and delivers the results to the requester.

All communication and cooperation between modules is achieved via messages expressed in the Common 'universal' Interagent Communication Language (ICL). ICL includes a conversation layer and content layer. The conversational layer is defined by event types together with the parameter lists associated with them.

On the other hand the content layer consists of the specific goals, triggers and data elements that may be embedded within various events. The content layer of ICL has been designed as an extension of the PROLOG programming language to take advantage of unification and other PROLOG features. Thus, compound goals can be expressed by using PROLOG-like operators and also parallel goals to be processed competitively or simultaneously by different modules. Furthermore conditional execution and constraints on executions can also be expressed. Altogether, OAA's ICL offers a very powerful communication mechanism.

2.4 Selecting OAA

The final decision to choose the Open Agent Architecture for the TALK project is mainly based on two factors: (i) the extensive previous experience that most partners have with OAA which will ensure maximum support for the platform within the TALK project and (ii) the corresponding existing work already based on OAA which gives us a head start with the work on connecting modules through the OAA architecture as described in the next section.

Chapter 3

Connecting Modules with Middleware Wrappers

This chapter gives an overview over the integration of components (modules) that are either being developed or are already available. The integration of components typically consists of a software interface layer that connects the component with the OAA architecture. This process is referred to as *wrapping* and requires additional coding and includes the identification of a relevant set of methods hat the component must provide to the system–the Application Programming Interface (API)–and a recasting of this API as a set of *solvables* that are presented to the OAA facilitator.

We therefore include in the short descriptions of the modules in this chapter the list of solvables where applicable. A much more comprehensive set of information about the modules can be found in the software questionnaire that is available separately. It includes authors, contact points, soft- and hardware requirements, runtime behavior, licensing information and more.

3.1 Software Questionnaire

Using a standardized questionnaire, we collected information about existing software at all partners. This has provided us with an overview over the requirements of this software in terms of operating systems, programming languages, etc., but also in terms of communication. The last point helped the choice of middleware that is described in section 2.

The items queried in the questionnaire are listed in the appendix, the complete results of the questionnaire were published internally. The document is updated when new information becomes available. The last version dates from November 5, 2004 and an update is currently being prepared.

3.2 Overview over Existing Modules

This list gives a quick overview over modules that have been connected to the OAA architecture and are described in the following:

- ASR: ATK

- Dipper
- 0-Plan
- USE: Voice Recognizer Manager, Synthesizer, Knowledge Manager, Home Setup, Dialogue Manager, Device Manager, Action Manager
- iCal
- jlGui
- ffmpeg
- Festival
- Free TTS
- Sphinx 4
- MapAgent
- GF
- Trindikit 3.2
- GoDiS
- RapidFire
- FreeDB
- Mary

In the following sections, partner names in square brackets at the beginning indicate which partner has contributed this module, in most cases by supplying the wrapping agent.

3.2.1 ATK Real-time Speech Recognition

[UCAM/UEDIN] ATK is 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.

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. 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

ATK is being used in the CMI ”SCILL” (computer assisted language learning) project. UEDIN is already using ATK and UGOT has expressed an interest in using ATK.

Links

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

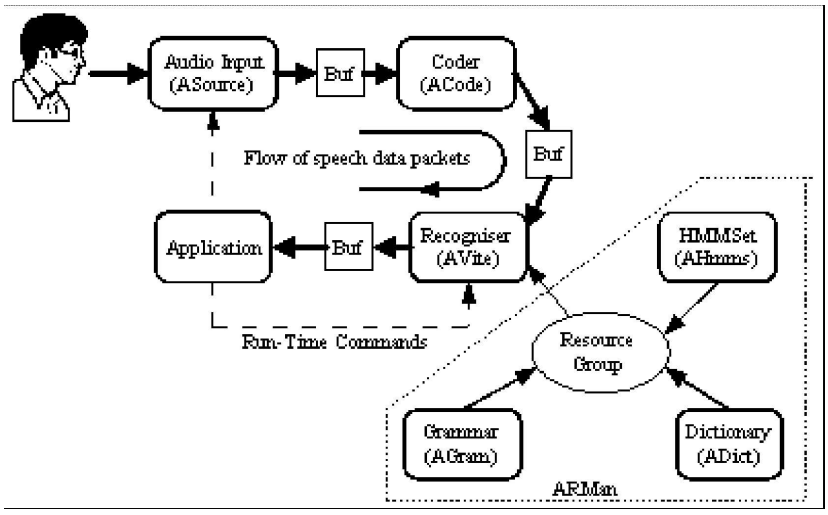


Figure 3.1: Sketch of the ATK internal architecture.

3.2.2 Dipper

[UEDIN] 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 [BKLO03] and [BO03].

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 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 version 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 3.2). 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 3.2: The DIPPER Graphical User Interface.

The current version of the DIPPER DME is implemented in Prolog. 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. In the Java version, Prolog calls go via OAA as well.

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 people from other research groups

too.

Links

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

3.2.3 O-Plan

[UEDIN] The O-Plan (Open-Planning) project began in 1983. In our work on dialogue systems we use O-Plan in several ways:

- planning sequences of dialogue moves (e.g. Ask origin, then ask destination)
- content planning (e.g. Tell the user about their top rated option)
- planning rhetorical structure (e.g. Contrast option1 and option2 in terms of attribute1)

O-Plan runs as an Allegro Common Lisp program (ACL 6.2). To run O-Plan the user must define the file `myplan.tf`, specifying all the plan operators for the domain.
It is used in the BEETLE tutorial dialogue system project at Edinburgh. It is used on numerous planning projects and applications (again see O-Plan main website)

Solvables

The interface to O-Plan is currently via the OAA Lisp layer (via backwards compatibility with OAA v1). A full description of its use in a dialogue system can be found in [MFLW04]: `http://homepages.inf.ed.ac.uk/olemon/flairs04.pdf`.
Inputs to O-Plan are via OAA solvables (e.g. Called from DIPPER) such as:
`deliberate('mytask','myplan.tf',_X)` initialisation of O-Plan with `'mytask'` as the initial task in task specification file `'myplan.tf'`
`execution_mode(_Y)` put O-Plan into execution mode (necessary for initialization)
`next_step(Step)` get the next step in the plan

Links

- See `http://www.aiai.ed.ac.uk/~oplan/` for downloads and a full description of O-Plan generally.

The O-Plan project is sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under grant numbers F30602-95-1-0022 and F30602-99-1-0024.

3.2.4 USE Dialogue System Components

[USE] At the University Seville, a whole set of components for a dialogue system is used that communicate through the OAA architecture. This section briefly describes all these components.

Our system is a distributed and multiplatform set of collaborative agents. The goal is to accomplish complex tasks by interacting with the user, processing the information through a dialogue manager based on the Information State Update model, and executing (if necessary) the tasks.

We use OAA's ICL to communicate with other agents. At this stage we don't use any markup language, neither for inter-agent communication nor for the Dialogue Moves or the Semantic Network. See also [QGS+01] and [QA02].

Voice Recognition Manager

This agent is in charge of getting the utterances from the speaker. We don't work at low level (neither with the signal features nor even with the lattice), we just take the first from the N-Best hypothesis.

The VRM provides the result from the ASR to the Dialogue Manager with hardly no delay. It only listens to the speaker when the Dialogue Manager tells it to do so.

The VRM Agent proposes one solvable: `vrGetSpeech`. When called by the Dialogue Manager, this solvable returns as solution the result from the ASR.

Synthesizer

This agent is in charge of translating from text to speech the utterances from the Dialogue Manager.

The Synthesizer receives the text from the Dialogue Manager and translates it to speech in real time.

The Synthetiser Agent has two solvables:

- `setLanguage Description(Language)`: This solvable is used to set the language that the TTS has to work with. Possible values for `Language` are: 'spanish' and 'english'.
- `sayText(Text)` This solvable is used to transmit to the TTS the text it has to transform to speech.

We use three TTS engines: festival (free software, running under Linux), Microsoft TTS (free software, under Windows), Atlas (commercial application, under Windows).

Knowledge Manager

This agent contains both the static (structure of the house, general ontology, ...) and dynamic (installation, configuration and status of the devices) knowledge involved in the task.

The Knowledge Manager is in charge of storing and organizing the information related to the devices installed in the house regarding location, characteristics and relations with other devices. The approach adopted for organizing this information is based on the conformation of a semantic network.

The Knowledge Manager functionalities are used when the system needs to get information from the ontology. In our present implementation, only the Action Manager request the solvables from the Knowledge Manager.

On the other hand, to accomplish with its functions, the Knowledge Manager needs the Device Manager in order the status of the devices.
As input to this module, the Knowledge Manager proposes twelve solvables:

- `kmCreateNode`: It evaluates if the a node with the same descriptor already exists. If not, a new Semantic Node is created. Parameter: Descriptor.
- `kmDependence` This solvable creates a new parent-child relationship between two Nodes. Parameters (2): Parent descriptor, child descriptor
- `kmInsertCompo` Adds a component to the list of components of a given node (identified by a descriptor). Parameters (3): Descriptor of the node, label and value of the new component.
- `kmAddDescriptor` Adds a descriptor to the list of descriptors of a given node (identified by a descriptor already in its list). Parameters (2): Target descriptor, new descriptor.
- `kmPrintSN` Prints the semantic network (SN) and the list of pairs descriptor-node (Descriptors) on the standard output. Parameters : None.
- `kmGetCompos` Makes a list of the components of the subtree of the semantic network, starting from the item with a given descriptor. Parameters (1): Descriptor.
- `kmDeviceResolution` Given a list of positive and negative descriptors (descriptors that define positive or exclude negative features of a certain device), this module finds in the SN the matching devices. The KM also checks the state of the devices by means of the solvable devmExecuteAction. Parameters (7): location, devtype, descriptor, devstate, exception_location, exception_devtype, exception_descriptor. The first four parameters describe features of the device (e.g : “the lamp in the bedroom”), and the other three defines features not present in the device (e.g. “all the lamps in the bedroom but the red ones”). All the parameters are optional.
- `kmGetCompos_union` Given a list of nodes, each identified by one of their descriptors, get the union of their components, and the components of their children. Parameters (1): Node list.
- `kmGetCompos_intersec` Given a list of nodes, each identified by one of their descriptors, the function returns the intersection of the lists resulting of a call to kmGetCompos(node_descriptor) for each given descriptor. Parameters (1): Node list.
- `kmLoadNetwork` Loads a Semantic Network. Parameters (1): File name.
- `kmClearNetwork` Frees the Semantic Network. Parameters: None.
- `kmRemoveAllCompo` Removes all the components in the network whose label match a given one. Parameters (1): Label.

Home Setup

The Home Setup is a graphical tool for simulating a home environment. It can work alone or coupled with physical devices (which should be controlled by the Device Manager). In addition to this agent we provide also the Home Designer, a tool for making the design of the house.

The Home Setup presents graphically the status of the house, indicating where are the devices and their status. It is a useful way to check that the light has actually be switched on when we have asked for it.
As input to this module, the Home Setup proposes seven solvables:

- `hsUpdateDeviceState` Updates the state of a given device. Parameters (2): Label, state. The first parameter identifies the device, the second parameter specifies its new status.
- `hsSw2On` Command "switch on". Parameters (1): Label. Identifies the device.
- `hsSw2Off` Command "switch off". Parameters (1): Label. Identifies the device.
- `hsSw2ISt` Solvable used to check the status of a given (not dimmable) device. Parameters (1): Label. Identifies the device.
- `hsSwdOn` Command used to update the bright level of a dimmer. Parameters (2): Label, level. The first parameter identifies the device. The second parameter adjusts the level of the dimmer.
- `hsSwdOff` Command used to switch off a dimmer. Parameters (1): Label. Identifies the device.
- `hsSwdISt` Solvable used to check the status of dimmer. Parameters (1): Label. Identifies the device.

Dialogue Manager

Our Dialogue Manager is composed by the following modules:

- Episteme: This is the main module of our system. It is a linguistic tool based on unification grammars. Its functionalities are:
  - lexical/morphological analysis
  - syntactic analysis (parsing)
  - unification
- Vtree: Module in charge of efficient storage and retrieval of large lexical databases.
- Mph: Module whose role is the automatic morphological generation based on linguistic knowledge.
- Delfos: Delfos is our implementation of the Information State Update paradigm for Natural Commands Language. We have formalized it through our so-called DTAC structures, whose components are the following:
  - DMOVE (Dialogue Move): Updates to be applied to the information state.
  - Type: We have made a sub-division of the Dialogue Moves. For instance, the Dialogue Move 'SpecifyCommand' may have different types such as 'CommandOn' or 'CommandOff'.
  - Args (Arguments): Within a Dialogue Move there may be some complex arguments. These arguments will be placed in this field, by means of nested DTAC structures. If a given Dialogue Move requires an Argument but it is empty, the Dialogue Manager will launch an expectation to fulfill this field. The arguments take the form of a list in which conjunction, disjunction and optional operators may appear.

  - Conts (Content): This feature represents the particular values associated to each element in the ARGS attribute. For terminal DTAC structures (i.e., with an empty ARGS list), the CONT will specify the value of the structure. For non-terminal DTAC structures (i.e. with non-empty ARGS list), the CONT is recursively represented by the CONT feature of each feature whose name equals the ARGS value.

- DM Agent: This module acts as interface with the rest of the system. It publishes the solvables to the OAA facilitator, and asks for functionalities to the rest of agents when needed.

The Dialogue Manager processes the utterances received from the Voice Recognition Agent in a real-time basis, and decides the following steps to perform correctly the requested task. This step may be the actual execution of a command, a question to the user asking for more information, accessing the Knowledge Database to check the status of a device.

The Dialogue Manager doesn't declare any solvable. The communication with other agents is always started by the Dialogue Manager.

Device Manager

The Device Manager is the interface of the system with the real devices. It links the virtual Home Setup with the physical addresses.

The Device Manager interacts with the physical devices by means of the X10 protocol. It is able to switch on/off, to adjust the level and to ask for the status of a given device. These functionalities are accessible by other agents through OAA.

As input to this module, the Device Manager proposes three solvables:

- `devmInit` Initializes the devices table. Parameters: None.
- `devmCatchDevice` Given a label, devtype and/or the physical address, this solvable finds the referred device, and updates its content. If it didn't exist previously, the Device Manager creates a new entry in the table with this device. Parameters (4): Label, devtype, physical address and state. These are the parameters of a new device or the new parameters of an existing one.
- `devmExecuteAction` Receive a command and execute it. Possible commands: state, on, off, dim. Parameters (3): Command, arguments, label. The first parameter specifies the command requested. The second parameter is the argument (actually the level if we are to deal with a dimmer). The third parameter is the label of the device.

Action Manager

The current devices will support the actual functionality of the system. This functionality (turn on/off the device, ask the current status, change the level of a dimmable device,...) must be accessed through the Home Setup agent, in charge of the simulation and monitoring of the device. Nevertheless the devices incorporate primitive functions to the system. The Action Manager offers new higher level functionality that will be translated into sequences of primitive actions.

The actual tasks to be executed are always transmitted through the Action Manager agent. The Dialogue Manager makes use of the Action Manager's functionalities. The Action Manager, on its side, must call both the Knowledge Manager and the Device Manager.

As input to this module, the Action Manager proposes two solvables (both to be used by the dialogue manager):

- `amDeviceResolution` This module finds in the Semantic Network the devices matching a given a list of positive and negative descriptors (descriptors that define -positive- or exclude -negative- features of a certain device). The operation is actually taken in charge by the Knowledge Manager, the Action Manager just transmit the request. Parameters (7): location, devtype, descriptor, devstate, exception_location, exception_devtype, exception_descriptor. The first four parameters describe features of the device (e.g., "the lamp in the bedroom"), and the other three defines features not present in the device (e.g., "all the lamps in the bedroom but the red ones"). All the parameters are optional.
- `amExecuteCommand` This solvable is used when an action is to be executed (e.g., switching on a light). The actual action is requested to the Device Manager by means of its solvable devmExecute-Action. Parameters (3): Command, label, arguments. The first parameter is the action requested (possible actions are: state, on, off, dim, set, bright). The second one identifies the device. The third parameter is optional and is used to ask for a precise level when the device is a dimmer.

3.2.5 iCal Agent

[UGOT] This is an OAA agent that parses and generates calendar files of the iCal format. At UGOT, it will be used to connect AgendaTALK with calendar applications using the iCal format.
The iCal agent offers two basic services: searching and adding iCal items.

Links

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

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 *YYYYMMDD*`T`*HHMM*, 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.

3.2.6 jlGui Wrapper Agent

[UGOT] 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.

Links

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

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.

3.2.7 *ffmpeg* Wrapper Agent

[UGOT] 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.

Links

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

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).

3.2.8 Festival Wrapper Agent

[UEDIN/UGOT] 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 agent in section 3.2.4.

Links

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

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.

3.2.9 FreeTTS Wrapper Agent

[UGOT] The FreeTTS Wrapper Agent connects FreeTTS to the OAA framework. It is possible to speak strings, files and urls, to change voice, and to use MBROLA voices.

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`

Solvables

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

3.2.10 Sphinx 4 Wrapper Agent

[UGOT] 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.

Links

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

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.

3.2.11 MapAgent

[UGOT] 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 a graph.

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`

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.

3.2.12 GF Agent

[UGOT] 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.

Links

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

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.

3.2.13 TrindiKit 3.2

[UGOT] TrindiKit is a toolkit for building and experimenting with dialogue move engines and information states, that has been developed in the TRINDI and SIRIDUS 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..

User Definable Knowledge Sources

If we take the “user” to mean the dialogue system designer, the user definable knowledge sources for a typical TrindiKit system would be IS type specification, dialogue moves, update rules, update and control algorithms, and additional modules and resources.

Description of the Input/Output Interfaces

A system built using the 3.2 version of TrindiKit can run as an OAA agent. However, it does not offer any services to the OAA community. Instead, it works as a “master agent” and uses other OAA agents

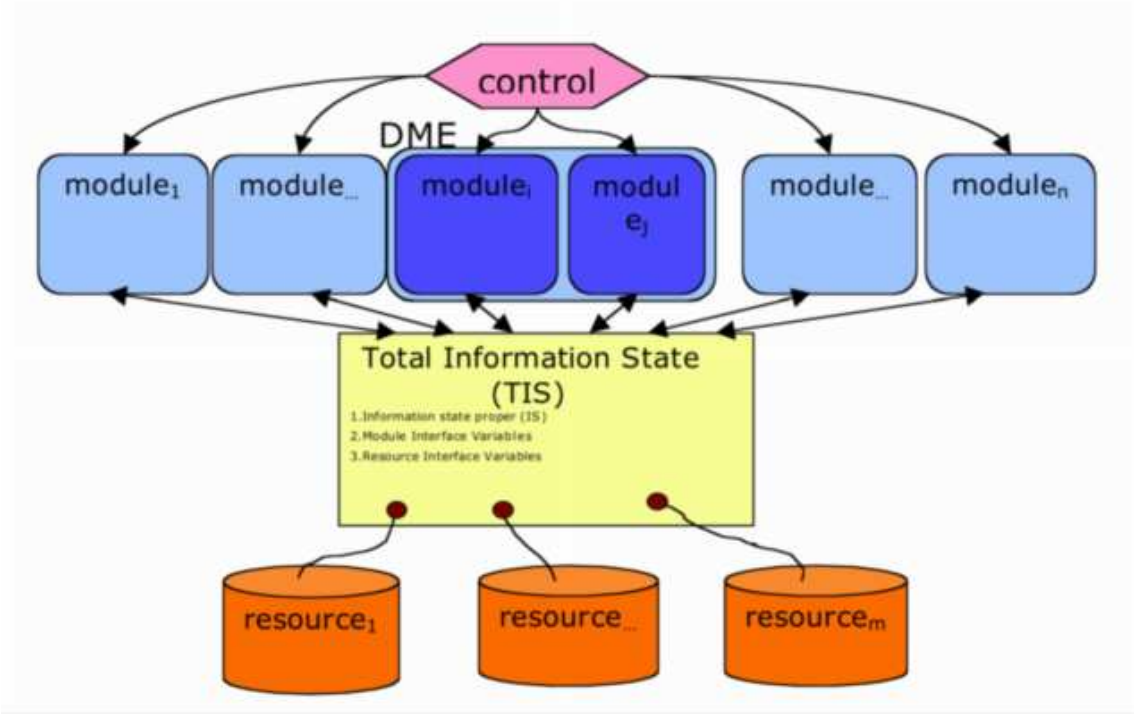


Figure 3.3: Schematic overview of TrindiKit

as modules or resources. The planned version 4 of TrindiKit will work differently, which means that a TrindiKit system will offer services to other OAA agents. It also means that the TrindiKit system is not necessarily the "master agent" but is instead called by some other agent. Planned OAA solvables include checking conditions, applying IS updates, and applying IS update rules.

Links

- `http://www.ling.gu.se/projekt/trindi/trindikit`

3.2.14 GoDiS

[UGOT] 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 [Lar02]. To implement a working GoDiS application one additionally needs to add resources (typically lexicon, domain knowledge, database or device interface) and modules for NL processing (typically, Nuance ASR and Vocaliser, a phrase-spotting interpretation module, and a template-based generation module).

There are two variants, GoDiS-IOD for inquiry oriented dialogue (e.g., database search) and GoDiS-AOD for action-oriented dialogue (e.g., controlling a device). The GoDiS implementation still contains some minor bugs but runs smoothly most of the time.

User Definable Knowledge Sources

If we take the "user" to mean the application designer, the user definable knowledge sources for a typical GoDiS-AOD application would be a lexicon/grammar, domain knowledge (including dialogue plans) and a device interface to connect some outside device to the system.

Links

- `http://www.ling.gu.se/grupper/dialoglab/godis/`

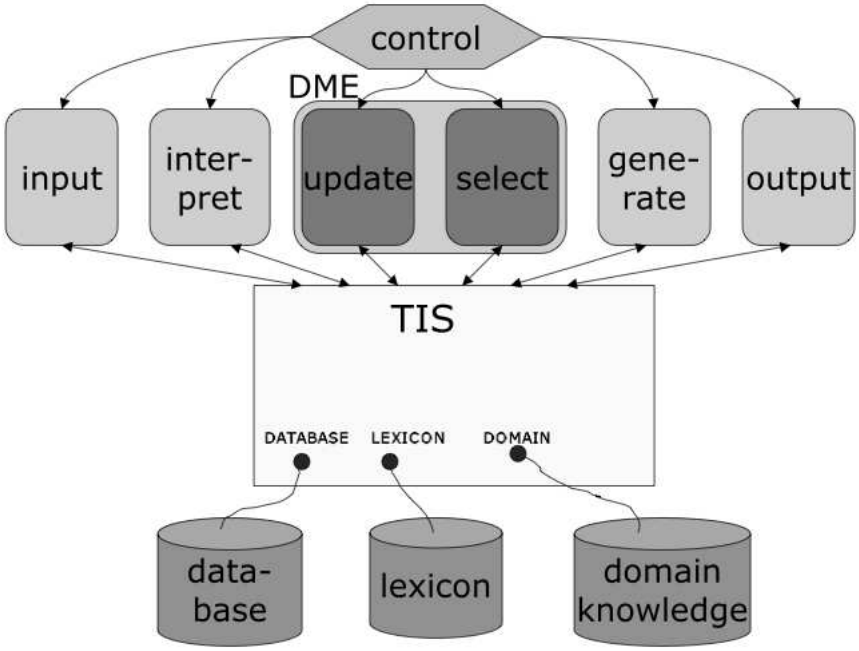


Figure 3.4: Schematic overview of a GoDiS application

3.2.15 Grammatical Framework (GF)

[UGOT] 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. TALK Deliverable D1.2a describes GF in full detail with many examples.

User Definable Knowledge Sources

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

Description of the Input/Output Interfaces

GF uses Unix stdio and a Java GUI.

Links

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

3.2.16 FreeDB

[CLT] 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 is currently being upgraded to allow playlist manipulation over OAA as well.

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.

Description of Input/Output Interfaces

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.

Use and Licenses

FreeDB Wizard was developed for TALK. It is also currently being used by students in a seminar on Dialogue Modeling at Saarland University.

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.

3.2.17 RapidFire

[USAAR] 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 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. (More details on RapidFire can be found in status report T3.3s1.)

User Definable Knowledge Sources

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

Description of Input/Output Interfaces

The RapidFire modules can either be used stand-alone with keyboard input (for testing) or 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.

Use and Licenses

RapidFire was developed for TALK. It is also currently being used by students in a seminar on Dialogue Modeling at Saarland University.

RapidFire is internally available to all TALK members. It is to be released next year as open-source software.

3.2.18 Mary

[DFKI] 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.

Links

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

Chapter 4

General Purpose Modules

This chapter describes finished and ongoing work on generic modules that are provided for use in multiple systems. Note that some of the modules already described in chapter 3 also fall more or less into this category. However, the complete modules described here, beyond the OAA wrapper, are specifically developed as generic modules within the TALK project. The list of modules encompasses:

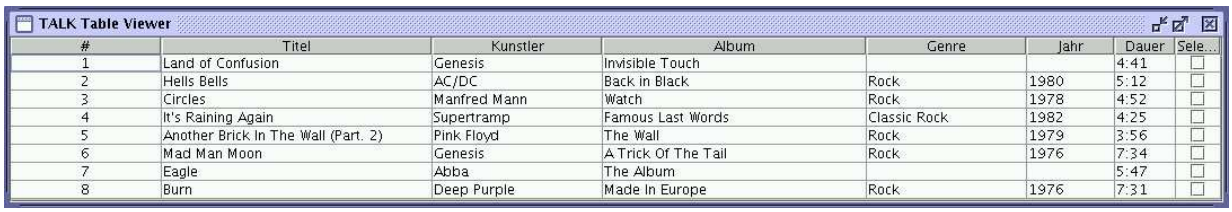
- Generic graphical display tool (Table Presenter)
- General database tool (MySQL)
- Pen/Mouse input tool
- Ergocommander
- Keyboard input tool for Wizard-of-Oz experiments

4.1 Table Presenter

The table presentation module from DFKI is a first step towards a generic graphical display module. The current 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 4.1 for a table with variable with columns), or instead simply display a text message. See figure 4.2 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 use 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 4.1: Screenshot from the table presenter module. It shows a table of MP3 songs to choose from.

4.1.1 Solvables

The two main solvables for the table presenter module are `show_table` and `show_list`. Some syntactical variants exist that are not listed here.

- `show_table(+TableHeader, +TableData)`
  - `TableHeader` - A string-encoded list of the names for the column heads.
  - `TableData` - A string-encoded list of data sets, i.e., rows. Each row is itself a list of data items and must have the same number of elements as the TableHeader.
- `show_list(+TableData)`
  - `TableData` - A string-encoded list of data sets, i.e., rows. Each row is itself a list of data items and all rows must have the same number of elements.
- `show_text(+Text)`
  - `Text` - The text message to be displayed in the window instead of a list or a table.

4.1.2 Options Presenter

For the Wizard-of-Oz experiments, we need 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 4.2), to mirror the user screen.

4.2 Database Module

Since a number of systems in TALK are dealing with database access, DFKI is planning to provide a generic tool that connects MySQL-based databases through OAA. CLT already has gathered experience from an initial version of the FreeDB tool that was based on MySQL and there is a JDBC-based SQL agent available from SRI. The current timeline plans completion in May 2005, but this might be moved to an earlier date if needed by TALK partners.

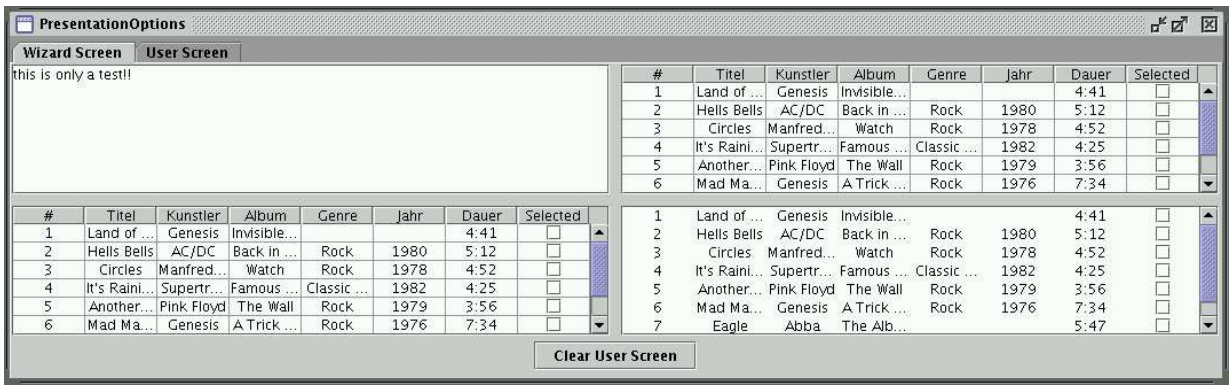


Figure 4.2: Screenshot from the options presenter module. It shows four different views for the wizard to select from: a simple text message, two tables and a list. Note that the list layout is currently being redesigned.

4.3 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. Figure 4.3 shows the rudimentary interface of the current prototype. We expect completion in May 2005, see also the timeline in chapter 6.1. An option would be the inclusion of functionality from the map agent by UGOT, as described in section 3.2.11.

4.4 Ergocommander

BMW has provided software components (and all necessary hardware) to use the ergocommander (similar to the iDrive) in the TALK project. This will enable us to test the in-car baseline system 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 and interface software that maps the movements of the input device to keyboard strokes.

This setup is currently being evaluated and will then be wrapped in an OAA agent to modularize the setup as much as possible. We can thus implement other parts of the input interface, e.g., graphical displays, independently from the actual input device, even simulating the ergocommander with a keyboard.

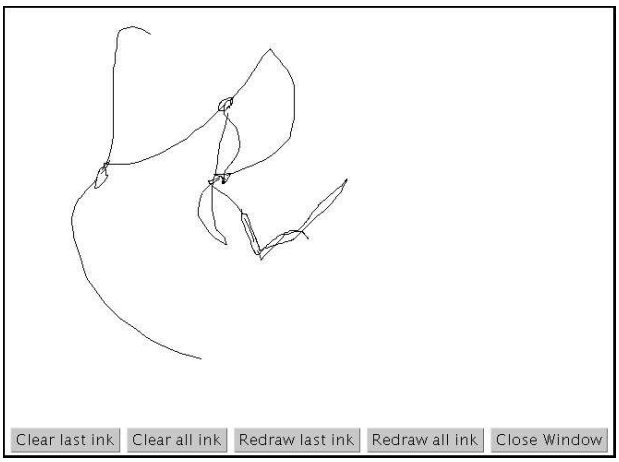


Figure 4.3: Unknown artist. Screenshot from the initial version of the pen input capturing module. As the buttons suggest, streams of input coordinates are grouped as strokes of 'ink'.

4.5 Keyboard Input Module

For a more specific purpose, namely the Wizard-of-Oz experiments at Saarbr¨ucken, 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 4.4 shows the interface during the spelling correction phase with two options (within the maximal edit-distance of 2) for the word "you".

Figure 4.4: 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.

Chapter 5

Systems

This chapter gives, as examples, a short description of two systems that we are working on in the TALK project, i.e., using the OAA middleware (see chapter 2) to connect modules as described in the last chapter. Status report T5.2s1 and the various status reports and deliverables for workpackages 1, 2, 3, and 4 include more comprehensive descriptions of these and other systems.

5.1 UEDIN-UCAM

The UEDIN-UCAM baseline dialogue system is built around the DIPPER dialogue manager [BKLO03] and will be used primarily to explore issues in the evaluation of hand-coded versus learned dialogue strategies.

5.1.1 System Components

Communication between components is handled by OAA's asynchronous hub architecture, and all components currently run under Linux. The major components are:

- ATK for speech recognition, see section 3.2.1.
- DIPPER for ISU-based dialogue management, see section 3.2.2
- Dialogue policy "advisor" agent[1]
- Multimodal Map interface (adapted from the UGOT Tram system), for multimodal input and output[2], see sections 3.2.11 and 4.3.
- GF parser agent, see sections 3.2.12 and 3.2.15.
- Festival for speech synthesis, see section 3.2.8.

[1] Uses dialogue management policies learned from data. This is written in Python and has an OAA wrapper in C

[2] This component is being developed by DFKI.

These components and their current state are described fully in deliverable D4.1. The current state of the system is that we are working on modifying an existing end-to-end dialogue system for flight information, using many of these components, to the in-car domain.

5.2 Wizard-of-Oz System

The USAAR-DFKI experimental system for Wizard-of-Oz (WoZ) experiments serves two purposes. Primarily, we aim to collect data on multimodal presentation strategies, concentrating on the presentation of large sets (lists) of data. The experiments will also provide a corpus of transcribed user and wizard utterances. Secondarily, a number of the components included in the WoZ system will be used in the in-car baseline system, developed in task 5.2.

5.2.1 System Architecture

All software components in the WoZ experiment (except for the Lane Change Task) are wrapped as OAA agents, running on at least four different machines in a mixed fashion under various versions of Linux and Windows. The major components are:

- Keyboard input module: one for the user, another for the wizard, see section 4.5
- FreeDB database of CD titles, see section 3.2.16
- Presentation planning agent that computes various presentation options for the
- Options presentation agent, see section 4.1.2 that send the selected presentation to the user's table presentation agent, see 4.1
- Mary speech synthesis agent, see section 3.2.18.

The status reports for workpackage 3 describe the system in more detail, the progress towards the in-car baseline system is reported in status report T5.2s1.

Chapter 6

Conclusion and Plans for 2005

In the first year of the TALK project, many parts of the basic infrastructure have been implemented. We have decided on the OAA architecture as the common software middleware in the TALK project. This will allow us in principle to connect modules from all partners, allowing an easy reuse of modules where applicable. We have also wrapped among the partners a number of existing software components to connect them as OAA agents. Work is ongoing on providing generic modules that can be reused in different applications.

Based on this setup, a number of basic systems are already in place at some partners that are now being extended to include the research results of TALK. The baseline in-car system to be completed in task 5.2, Integration in June 2005 will also be based on the architecture described here; a number of components are already in place, more are currently under development.

6.1 Timeline

The following list briefly summarizes the steps planned for Task 5.1 Infrastructure during the next year. In the first year, many parts of the basic infrastructure have been implemented, so work now concentrates of adding missing modules, providing further generic modules and support the systems integration. The latter is done, e.g., by providing support for logging system data and extracting (filtering) information from logfiles in various formats, e.g., for analysis of the Wizard-of-Oz experiments.

- Adding OAA wrappers to further modules [all partners as needed]
- OAA wrapper for Ergocommander. [DFKI, Mar 05]
- Generic database tool as OAA module. [DFKI, May 05]
- OAA wrapped and enhanced version of generic pen input tool. [DFKI, May 05]
- Generic graphical presentation module (updated version of table presenter). [DFKI, May05]
- Support for in-car baseline system. [USAAR, DFKI, BMW, Jun 05]
- Enhanced OAA logging support [DFKI, Jun and Dec 05]

- Tools for logfile filtering [DFKI, Jun and Dec 05]
- Combining MapAgent with pen input tool [DFKI, Aug 05]
- 2005 status report on Task 5.1 [DFKI, Dec 05]

Bibliography

[BKLO03] Johan Bos, Ewan Klein, Oliver Lemon, and Tetsushi Oka. Dipper: Description and formalisation of an information-state update dialogue system architecture. In *4th SIGdial Workshop on Discourse and Dialogue*, Sapporo, Japan, 2003.

[BO03] 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.

[HNM+04] Gerd Herzog, Alassane Ndiaye, Stefan Merten, Heinz Kirchmann, Tilman Becker, and Peter Poller. Large-scale software integration for spoken language and multimodal dialog systems. *Journal Natural Language Engineering*, 10(3–4):283–305, September 2004.

[Lar02] Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, University Gothenburg, 2002.

[MCM99] D. L. Martin, A. J. Cheyer, and D. B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence: An International Journal*, 13(1-2):91–128, January-March 1999.

[MFLW04] Johanna D. Moore, Mary Ellen Foster, Oliver Lemon, and Michael White. Generating tailored, comparative descriptions in spoken dialogue. In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Symposium Conference (FLAIRS)*, Miami Beach, Florida, August 2004. AAAI Press.

[Pol02] Peter Poller. State of the art in system architectures. Technical Report COMIC Deliverable D1.1, DFKI, 2002.

[QA02] Jose F. Quesada and J. Gabriel Amores. Knowledge-based reference resolution for dialogue management in a home domain environment. In Mary Ellen Foster Johan Bos and Colin Matheson, editors, *Proceedings of the sixth workshop on the semantics and pragmatics of dialogue (Edilog)*, pages 149–154, Edinburgh, September 2002.

[QGS+01] Jose F. Quesada, Federico Garcia, Ester Sena, Jose Angel Bernal, and Gabriel Amores. Dialogue management in a home machine environment: Linguistic components over an agent architecture. *SEPLN*, 27:89–98, September 2001.

Appendix

This software questionnaire was completed by all partners to provide an overview over existing software and collect requirements for a number of purposes, as outlined in section 3.1.

- 1. General Information:
  - Author of this description:
  - E-mail of Author:
  - Component/Software Name: (for a unique component identification, also a compound name, e.g., generator.text.german)
  - Workpackage(s):(the workpackage(s) your component is associated with)
  - Authors/Owner of the Component:
  - Technical Contact/Responsible Person:(preferably one person)
- 2. Technical Information:
  - Description of the Component Functionalities: (describe the component functionalities/tasks in a few sentences, including subcomponents/Data exchange diagrams and also addressing the current state of implementation)
  - Runtime Behaviour:
  - User Definable Knowledge Sources:
  - Description of the Input/Output Interfaces:
  - Used in other projects/by other groups:
- 3. System Requirements:
  - Programming Language(s):(planned components may have different implementation options)
  - Operating System(s):(perhaps more than one)
  - Size of the Component (Source or Binary Code):
  - Memory Requirements:
  - External Software needed: (special software requirements)
  - Hardware Requirements:(please describe used hardware devices (e.g., microphone types) as well as planned hardware that is needed or you'd prefer to use in conjunction with your component)

- 4. Description of Expected/Intended Interaction(s)/Interface(s) with/to other TALK Components:(please try to identify interfaces as far as possible at present)
  - (A) Input: (please identify input interface(s) of your component that you expect to be supplied by other components)
    - Interface 1:
      - Data Producer:
      - Data Content:
      - Data Format:
    - Interface 2: ...
  - (B) Output:(please identify output interface(s) of your component that you expect/imagine to be consumed by other components)
    - Interface 1
      - Recipient:
      - Data Content:
      - Data Format:
    - Interface 2 ...
  - (C) Technical Interface (please identify existing/preferred technical interface(s) of your component, e.g., RPC, OAA, ...)
    - Interface 1:
    - Interface 2: ...
  - (D) Required Knowledge Sources:(please try to identify (external) knowledge sources that are essential for your component)
    - Knowledge Source 1
      - Content/Description:
      - Producer:
      - Dynamic or Static: (y/n)
    - Knowledge Source 2 ...
- License/Distribution Conditions:
- Literature: (documentations/citations)
- Miscellaneous Information/Further Remarks: