D2.2 - Unified integration framework for different software modules

Project arconym:	ALIAS
Project name:	Adaptable Ambient Living Assistant
Strategic Objective:	ICT based solutions for Advancement of Social Interaction of Elderly People
Project number:	AAL-2009-2-049
Project Duration:	July, 1st 2010 – Juni, 30th 2013 (36months)
Co-ordinator:	Prof. Dr. Frank Wallhoff
Partners:	Technische Universität München Technische Universität Ilmenau Metralabs GmbH Cognesys GmbH EURECOM Guger Technologies Fraunhofer Gesellschaft pme Familenservice

D2.2

Version:	1.0
Date:	2011-12-23
Author:	FhG
Dissemination status:	PP

This project is co-funded by the Ambient Assisted Living (AAL) Joint programme, by the German BMBF, the French ANR, the Austrian BMVIT.

Once completed please e-mail to WP leader with a copy to
eric.bourguignon@tum.de and **frank@wallhoff.de.**

Del 2.2	Executive Summary
This document provides an overview of the ALIAS robot's unified integration framework containing the graphical user interface (GUI) and its integrated software modules. It outlines the methods of integration that have been applied in order to realize the unified software framework. The current version of the ALIAS robot's GUI includes a working Skype telephone functionality, a selection of games, a web-browser, and a virtual keyboard for the robot's touch-screen. Some of these modules have been integrated at source code level while some are running as separate processes, controlled by means of operating system functions.	

Dissemination Level of this deliverable (*Source: Alias Technical Annex p20 & 22***)**	
PP	Restricted to other programme participants (including the Commission Services).
Nature of this deliverable (*Source: Alias Technical Annex p20 & 22***)**	
P&R	Prototype and report

Due date of deliverable	M15
Actual submission date	20.12.2011
Evidence of delivery	23.12.2011

Authorisation			
No.	**Action**	**Company/Name**	**Date**
1	Prepared	FhG	15.12.2011
2	Approved	TUM	16.12.2011
3	Released	Cognesys	23.12.2011

Table of Contents

1. Introduction

This section presents a general overview of the robot's modules, relevant for this deliverable. Section 2 provides an overview of the graphical user interface (GUI) and the general concept on handling user inputs.

Section 3 takes a closer view on the integrated software modules and the techniques that have been applied to do so. Section 4 focuses on the GUI's interface to the Dialogue Manager, the supported commands, and how it can be remote-controlled by the robot's other modules.

Section 5 contains a short summery of this document.

The Robot's Modules

Figure 1 shows the general overview of the robots modules, though the focus of this deliverable will be on the graphical user interface (GUI,) its integrated components (e.g. Skype,) and its interfaces to internal and external modules, likewise. The figure also shows the Brain Computer Interface module, which is currently being integrated into the robot's system.

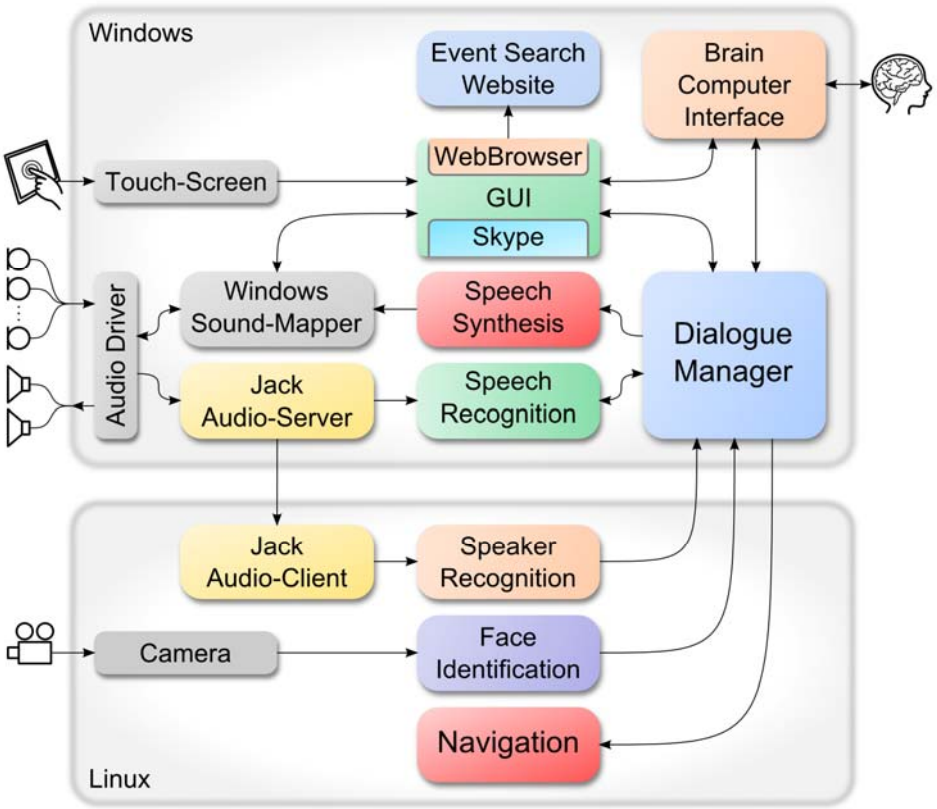


Figure 1: Overview of the ALIAS robot's modules, distributed on two computers.

The robot contains two separate computers, one running Windows 7 the other one running Fedora 14 Linux. The robot's software modules run distributed on both computers (cf. Figure 1,) while they are connected inside a local area network. The Dialogue Manager (DM) module represents the robot's most central component. It runs on the Windows computer and maintains connections to nearly all the remaining modules. The DM collects inputs and events from all these modules, interprets them and decides which actions to perform, i.e. commands to send to which modules.

The GUI runs on the Windows computer; it is one of the modules, connected to the Dialogue Manager. It integrates several applications and receives user inputs from the Windows computer's operating system. All of which is encapsulated to a single GUI module using a single connection to the Dialogue Manager.

2. The Graphical User Interface

This section provides an overview of the graphical user interface structure. The GUI consists of a series of menus containing few large buttons, each of them leading to another menu or starting an application, i.e. an integrated software module. The GUI's main menu is shown in Figure 1. The menu contains three buttons: leading to the games sub-menu, starting up the integrated telephone application, or the integrated web-browser.

Figure 2: The GUI's main menu. The Button "Spiele" brings-up the games sub-menu, while "Telefon" starts the telephone application, and "Web Browser" runs the integrated web-browser.

The GUI makes a clear distinction between menus and application modules, though both are supposed to look quite similar on the screen. Menus provide access to sub-menus and integrated software modules i.e. "applications," using a tree-like menu structure which is defined by a configuration file. Every menu is identified and accessed by a unique ID.

Application modules (e.g. like the Solitaire game) are also identified and accessed by means of unique IDs. But once an application has been started, there is no common structure. Every application implements its own layout, buttons, features, and remote-control capabilities. So some features are available after the application has been started, only.

2.1. Integration of Software Modules

Each integrated component is perceived as a separate program or an application to be executed within the GUI. There are different ways for integration of new modules into the GUI which are presented below. But there's no general solution for integrating features and components into the common user interface. The decision on which option to take must be made for every single component. In ALIAS all three integration methods described in the following have been used.

2.1.1. Self-Made Software Solutions

One option is to develop a full-blown software solution for every desired feature, e.g. an e-mail client or a web browser. This would result in a near optimal performance in terms of user experience, since every aspect of the program could be controlled. However the necessary workload to realize all these functions would be way beyond the scope of this project.

2.1.2. Third-Party Software

Another option would be to use existing third-party software and integrate it into the common user interface. Although previous “proof of concept” trials were promising, this course of action proved to be more cumbersome than expected.

Only very few programs integrate nicely, e.g. offer an API for integration to other programs. There is no common ground on the integration of third party software. So every single software application (and revision) requires special attention and is likely to break the common design. Some components even appear to be mutual exclusive, or tend to behave erratic when combined.

2.1.3. Public Domain Source Code

Use open source programs that are available in the public domain. This method is very similar to using self-developed software solutions, though third party source code needs to be adapted for use with the overall GUI. In some cases the modifications to the source code may be extensive and there may license issues for open source software. Finding suitable public domain sources turned out to be hard or unfeasible.

2.2. User Inputs

The GUI is able to process two kinds of user inputs; direct inputs and indirect inputs. Both input types will be further outlined below. This is relevant for interactions between the GUI and other robot modules.

2.2.1. Direct Inputs

The GUI accepts ordinary user inputs, as they are provided by the host computer’s operating system. In case of the ALIAS robot the main source of such inputs will be the touch screen. These inputs are considered as direct inputs, since they’re provided by the computer’s operation system and are handled by the GUI directly.

More generally every input falls into the group of direct inputs if the GUI is the first of the robot’s components in line to receive it. Accordingly even an incoming phone call is a direct input, because it’s triggered by an integrated GUI module. So, unless properly handled and propagated, no other module would ever know about it. Thus most direct inputs will be relayed to the Dialogue Manager to be interpreted and translated to remote commands.

2.2.2. Indirect Inputs

A second variant of user inputs is represented by the group of indirect inputs. Indirect inputs are network messages, received by the GUI. Basically indirect inputs are inputs that are handled by another module, but require a reaction by the GUI. Typically such indirect inputs are generated by the Dialogue Manager, as response to a speech input for example.

The user may issue a verbal command to the robot: “Call Britta, please!” The sound wave is picked up by the robot’s microphones, converted into a sampled audio signal which is redirected by the Jack Audio Server to the Speech Recognition module. The Speech Recognition module converts the audio signal to a textual representation which will be interpreted and processed by the Dialogue Manager. In case the Dialogue Manager finds a contact named “Britta” in its data base, it sends a series of network messages to the GUI, containing the required commands to bring up the telephone application and initiate the phone call.

As stated before, most direct user inputs to the GUI will be relayed to the Dialogue Manager as well. This ensures that the Dialogue Manager will be up-to-date at all time and may even intervene in case it deems the user input for improper, e.g. calling a contact who is not available at the moment.

In this case the Dialogue Manager is aware of the intent of calling the contact and may decide to remind the user as soon the requested contact is available, again. A more detailed description of the interface protocol is presented in section 4.

3. Integrated Components

The GUI acts as a host for multiple integrated components. This section provides an overview of the components, currently integrated. It also describes how the separate components have been implemented and addresses some of the difficulties that need to be considered for the future development and extension of the unified system.

3.1. Skype

The (video) telephone functionality is a very important aspect for the ALIAS project since it is common way of long distance communication. In order to achieve the highest user acceptance it is advisable to incorporate an already well-established telephone system instead of developing a new ALIAS-specific solution. In this area the Skype communication system seems the most viable option. And in addition, Skype offers APIs for integration to external applications like the ALIAS GUI. Basically there are two different Skype APIs, the Desktop API and the SkypeKit API, supporting varying levels and modes of integration.

Figure 3 shows the telephone screen as it had been used during the ALIAS field trials in September 2011 in Berlin.

Figure 3: The integrated telephone module as used during the ALIAS field trials in September 2011. The button "Anrufen" initiates the phone call, while "Zurück" switches back to the main menu. The dummy label "Telefonpartner" (engl. phone contact) would have been replaced by the current phone contact's name.

3.1.1. Skype Desktop API

The Skype Desktop API (formerly known as the Skype Public API) offers access to the currently running Skype application, which can be remote-controlled to perform the desired functions. The Desktop API is represented by an asynchronous text-based message system, using the Windows API in order to exchange text strings between the ALIAS robot's GUI and the actual Skype application.

The following fragment represents the initiation of a phone call, whereas a more detailed documentation of the Skype Desktop API is available on the Skype Developer website [1]. Figure 4 contains a short log of messages, required in order to try placing a Skype call to the Skype Testing Service. Each line represents a single message that has been sent or retrieved via the Windows API message handling system. This interface is asynchronous. Affiliated messages are connected by means of common IDs. For example, line "#23 CALL echo123" uses command ID "23" to initiate a call to the Skype user ID "echo123". The response from the Skype application "#23 CALL 2381 STATUS UNPLACED" uses the same command ID "23" to mark its affiliation to the original command, and it provides the client with the actual call ID "2381" and status.

```
#951 SEARCH FRIENDS
#951 USERS echo123, alias-jade
CONNSTATUS ONLINE
CURRENTUSERHANDLE sven.alias
USERSTATUS ONLINE
#23 CALL echo123
#23 CALL 2381 STATUS UNPLACED
CALL 2381 STATUS UNPLACED
CALL 2381 STATUS ROUTING
MESSAGE 2409 STATUS SENT
#40 ALTER CALL 2381 HANGUP
#40 ALTER CALL 2381 HANGUP
CALL 2381 STATUS MISSED
MESSAGE 2441 STATUS RECEIVED
```

Figure 4: Exemplary message exchange between the robot's GUI and the Skype application in order to try calling the Skype Test-Call Service.

Currently, the GUI includes a working "telephone" implementation using the Skype Desktop API. Whereas the remote-controlled Skype application will be hidden below the robot's GUI. It provides access to the list of contacts and provides free internet-based Skype calls or pay-calls to the regular telephone network. A video telephony is not yet implemented but is subject to current work. It should be possible to apply the same technique for the Skype video frame as for the games integration (see section 3.3.1).

[1] Skype Desktop API Reference Manual, http://developer.skype.com/public-api-reference

During the setup of the recent ALIAS field trials in Berlin (September 2011), handling the Skype application's several pop-up windows, i.e. keeping them from interfering with the robot's GUI, deemed to be annoying. Thus, the Skype Desktop API appears to be not suitable for real-world deployment and will not be used for the ALIAS robot in the future.

3.1.2. SkypeKit

The SkypeKit for Desktop API ("SkypeKit" for short) is a complete Software Development Kit (SDK) for developing Skype-like stand-alone applications running independently from the original Skype software. Access to the SkypeKit is more restrictive than to the Desktop API. As of the end of October 2011 support for video telephony has been added to the Windows version of the SkypeKit[2]. So using the SkypeKit 4.02 SDK seems a viable option, now.

A new version of the Telephone using the SkypeKit is currently under development.

3.2. Games

The graphical user interface contains a small selection of games, which is accessible via the games menu, see Figure 5. Every game is represented as an internal application object, which can be - once initialized - displayed or hidden at command.

The games section represents a kind of a testing area for different approaches in integrating external third-party applications. With this premise, and the presumption of later users being casual gamers, a few common games have been selected for integration. This section takes a closer view on the integrated games and describes some aspects of the applied techniques for integration.

Figure 5: The games menu which provides access to the games Chess, Sudoku, Tic-Tac-Toe, and Solitaire.

[2] Announcing SkypeKit for Desktop with Video Calling, http://blogs.skype.com/developer/2011/10/skypekit_for_desktop_video_calling.html

As for now games may be started by the Dialogue Manager (e.g. by speech commands) but not actually played using indirect inputs (see section 2.2.2). This is due to the lack of control over third-party game implementations and because of the lack of expert knowledge. Meaning the Dialogue Manager would need to know the game's rules, the current state of play, and common play tactics in order to interpret the user's inputs correctly. Currently the GUI doesn't feature any commands to play the available games. However games may be started and played by using the robot's touch screen, i.e. direct inputs (see section 2.2.1).

3.3.1. Chess

The chess game uses the *Titans Chess* which is part of the robot's pre-installed Microsoft Windows 7 operating system. The game consists of a large main window with a menu bar, see Figure 6. The game's menu allows access to several smaller windows, e.g. a configuration dialogue or an "about box."

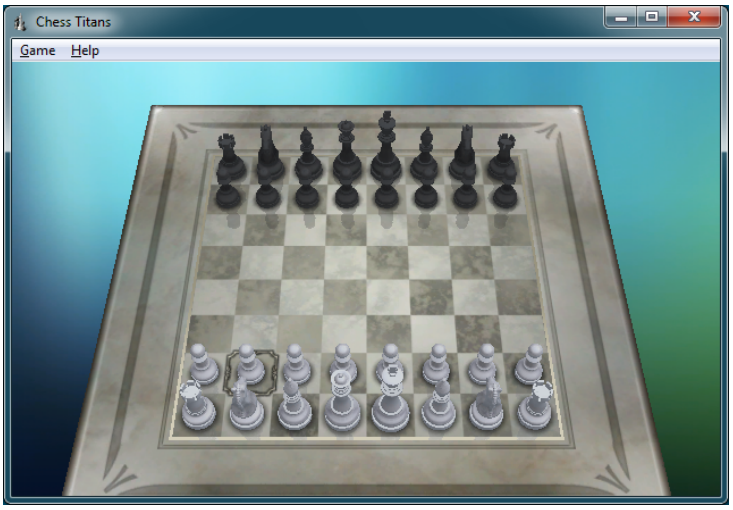


Figure 6: The original Titans Chess' main window.

Initial proof-of-concept attempts of using Windows API functions to reconfigure the game's main window to be a regular child module of the robot's GUI proved to be unreliable. This appears to be related to some internal buffers only being accessible to the process they originally have been created in. Since this procedure included moving the game off its original process into the robot GUI's process it lost connection to several internal modules and stopped handling user inputs. Though this process worked fairly well on another development system but not on the robot. The reasons for that are suspected to be timing issues during internal buffer allocations.

The current integration of the Titans Chess game is achieved by removing window frame and menu bar from the original game window by means of Windows API functions. Doing so, the first step is to retrieve the handle of the game's main window by means of Windows API functions. Then the game window's properties are changed to those of a Top-Most Window in order to make it hover above the robot's GUI application. Additional Windows API functions enable resizing and repositioning of the game window according to the desired specifications.

This makes the original games window look like an integrated part of the robot's GUI. Unfortunately this also renders the game's original menus inaccessible. However, keyboard short-cuts for all essential operations are still operational and will be emulated by the robot's GUI.

Figure 7 shows a screenshot of the resulting "integrated" chess application. Please note that the Windows Taskbar will be displayed on-top of the robot's GUI for as long the chess game is active, so it needs to be removed by means of additional tools.

Figure 7: The Titans Chess game, integrated to the robot's GUI.

3.3.2. Sudoku

The game Sudoku follows a different method of integration. It is completely integrated part of the GUI's source code, based on a freely available open source project[3]. It is depicted in Figure 8.

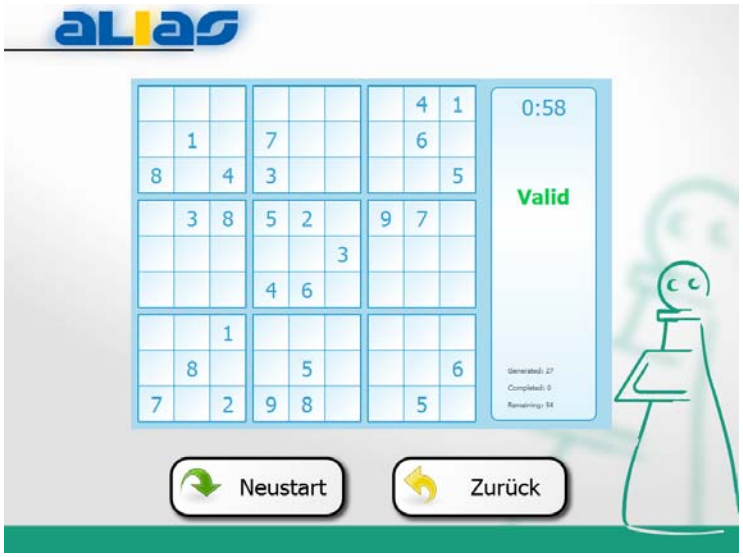


Figure 8: The Sudoku game screen.

[3] The source code of Wim Leers' Sudoku is available on the internet at http://github.com/wimleers/sudoku .

The original Sudoku program relied on mouse hover effects to implement the user inputs, i.e. a mouse cursor hover on a game field displayed a number selection dialogue within the actual game field. For use with the robot's touch screen the original number selection dialogue was too tiny. Mouse-hover events are not suitable for the robot's touch screen, also.

The availability of the game's source code made it possible to apply some modifications. So the mouse-hover event has been changed to a finger tip on the games field at question and the number selection dialogue has been simplified and enlarged to accommodate the requirements for the robot's touch screen. The modified input dialogue is shown in Figure 9.

Due to source code availability, all the game's functions (e.g. restarting a new game) are accessible and can be included in a new menu.

Figure 9: The Sudoku game's user input menu.

3.3.3. Solitaire

The Solitaire games is quite similar to the Titans Chess game (see section 3.3.1), it is also part of the robot's pre-installed Windows 7 operating system. But unlike the Titans Chess application, there are different versions of Solitaire on different Windows systems. While the re-parenting[4] approach, as mentioned in section 3.3.1, worked with the Windows XP Solitaire perfectly, it failed with the Windows 7 version.

Again, the original Solitaire application consists of a large main window, containing a menu bar and a status bar, see Figure 10. The menu bar offers access to additional dialogue windows, e.g. game configuration.

[4] Hierarchies in computer systems are often represented by parent-child relations, whereas the children (e.g. buttons) are the property of their parent (e.g. a dialog window). So making one window the property of another window corresponds to assigning parent and child roles, hence changing the parent of the designated child window. This process is commonly called re-parenting.

Figure 10: The original Windows 7 Solitaire's main window.

As before (section 3.3.1) the menu bar and window frame are removed by means of Windows API functions. The status bar is kept because it provides additional game information. The Solitaire window's properties are altered in order for it to become a top-most window. The robot's GUI incorporates an empty "dummy window" object, which is used to assess the appropriate size and position of the modified Solitaire window. Adjustments in size and location are applied using Windows API functions, again. The integrated Solitaire game is depicted in Figure 11.

Figure 11: The robot's integrated Solitaire game.

To provide consistent show and hide animations for the "integrated" Solitaire application, the previously mentioned "dummy window" object will be replaced by a picture of the current game window. In case the robot's Solitaire game is destined to hide, it will become invisible right away. The "dummy window" provides a substitute which can be animated, hence providing the illusion of the original game contents sliding off the screen.

3.3.4. Tic-Tac-Toe

The Tic-Tac-Toe game has been programmed from scratch, so its source code is also available and fully integrated to the robot's GUI. This is a simple game to be played by two users against each other, using a single robot or two robots that are connected e.g. via WiFi.

Figure 12: The Tic-Tac-Toe game screen.

The Tic-Tac-Toe game is pretty much the same as the Sudoku games except it extents the process by using a network interface.

3.3. Web-Browser

The web browser represents another essential part of the ALIAS robot's GUI. It provides a platform for the web-based services, like the Event Search Engine. During the early stages of the GUI's development it was decided to use the very popular third party web-browser Firefox[5] instead of developing a new web-browser solution from scratch. Since Firefox is a free open source project, there are lots of related projects and extensions. So Firefox is likely to "be around" and up-to-date for the foreseeable future.

There have been several attempts using different techniques to achieve proper web-browser integration. But so far there doesn't seem to be an optimal solution. All methods have their very own flaws and side effects that need to be taken into consideration.

A possible alternative is QtWebKit, which is the web-browser engine that comes with the Qt framework. So QtWebKit is already part of the GUI's development environment.

[5] Firefox, Mozilla Foundation http://www.mozilla.org/

3.3.1. Firefox and Microsoft Internet Explorer

Integration of the designated Firefox web browser proved to be cumbersome. The "proof of concept" implementation was not stable and the recently changed update policy is likely to break integration every few months. Though the Firefox source code is available, it is a vast software project. So processing and adapting its whole source code is beyond the scope of the ALIAS project.

The original proof of concept used to grab an existing Firefox window and re-parenting it to the robot's GUI worked well for displaying single web-pages. But navigation on these web-pages was prone to causing segmentation faults due to invalidated memory buffers. With the beginning of 2011, Firefox switched to a newer – more frequent – update policy. As for now this invalidated most aspects of integration for nearly every new release, every few months.

Another attempt was made by shifting the actual browser window off the computer screen, while displaying an image of it (without menus) on the robot's GUI. User inputs (e.g. mouse clicks or keyboard input) on the image then were relayed to the original browser application by means of windows API functions. However doing so caused trouble in handling focus events, so some control elements became inaccessible to the user. In addition frequent image updates (taking a picture, transferring and displaying it) cause a considerable system load. Some browser plug-ins used a similar procedure to display contents by drawing them atop the actual browser window – such contents were not part of the actual browser window and hence could not be displayed on the robot's GUI.

The windows version of Qt framework has the ability to integrate ActiveX containers into the GUI. Based on an integration attempt of the Microsoft Internet Explorer web-browser, a third party ActiveX container was used to encapsulate the Firefox's web-browser engine "Gecko." At first glance this worked nicely, but it induced some profound changes to the internal GUI configuration. Once displayed, the ActiveX container disabled the Qt framework's internal rendering engine and forced the entire GUI application to use native Windows rendering which caused flickering[6]. Hence animations became distorted and unpleasing. In addition there were focus handling issues; if the container lost the focus, none of its elements (e.g. a search dialog) was able to reacquire the focus again. Using browser plug-ins was supported by this method but cause the embedded browser to crash, frequently. The Internet Explorer performed better regarding the use of plug-ins, but was basically subject to the same issues as the Firefox. Support for this method of integration has been removed from the more recent Firefox versions 4 or newer.

Other than for the robot's GUI as acting as a mere program starter for the original stand-alone version of the Firefox web-browser, integrating a properly maintained Firefox doesn't seem a viable option, anymore.

[6] Graphical user interfaces consist of multiple elements that are placed atop of each other. For example, take a dialogue window containing a background, two buttons, and a frame around it. In case the display of this dialogue needs to be refreshed it'll be rendered one element after another: step 1 Background, step 2 frame, step 3 buttons. With native Windows rendering, each of these steps will be visible, causing flickering. The default Qt rendering uses a process called "back-buffering" in order to avoid flickering.

3.3.2. QtWebKit

The Qt framework which is used for development of the robot's GUI features a web-browser engine of its own: QtWebKit. It is based in the open source project WebKit[7], which is used by Apple's Safari or Google's Chrome browser, also.

Since QtWebKit is already part of the Qt framework it should be possible to integrate it into the robot's GUI. But instead of a full-blown web-browser, QtWebKit is an engine only. So a lot of common browser features (e.g. bookmarks or geo-location) are not directly available and need to be implemented. Figure 13 shows the current web-browser implementation, using the QtWebKit.

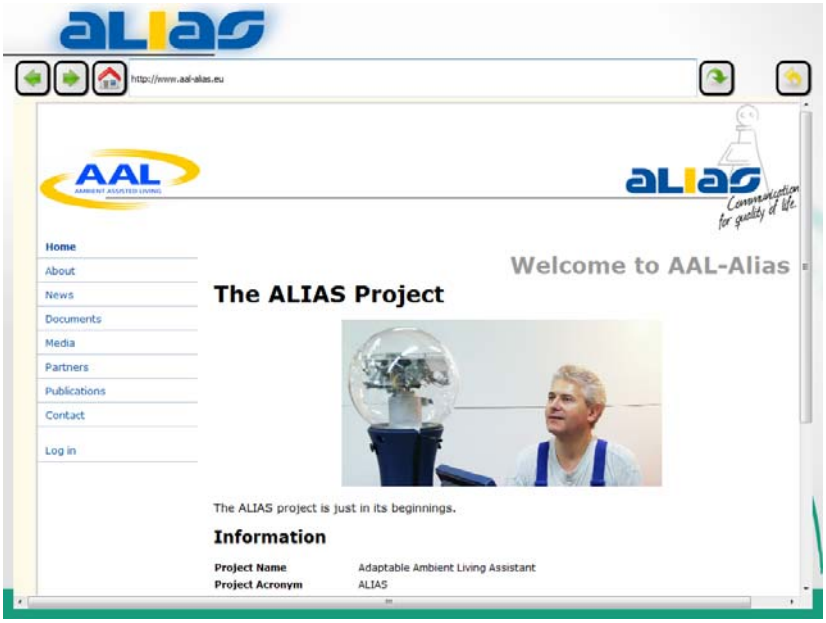


Figure 13: The current web-browser implementation, using QtWebKit.

The web-browser being a work in progress, still, is not connected to the Dialogue Manager, yet. Also its button layout, design and font sizes represent the initial draft and are not final.

3.4. Virtual Keyboard

As with the addition of the web-browser module the means of providing keyboard input via the touch-screen became essential. Hence previous approaches of integrating the virtual keyboard have been joined with the robot's GUI. As decided before, the third party software Hot Virtual Keyboard[8] is used to provide the actual virtual keyboard implementation. Figure 14 and Figure 15 show the current integration of the Hot Virtual Keyboard, which has been redesigned to accommodate the robot's GUI, though integration and design is still a work in progress.

[7] The WebKit Open Source Project, http://www.webkit.org/
[8] Hot Virtual Keyboard, http://hot-virtual-keyboard.com

Figure 14: In the top-right corner a button has been added to enable the virtual keyboard.

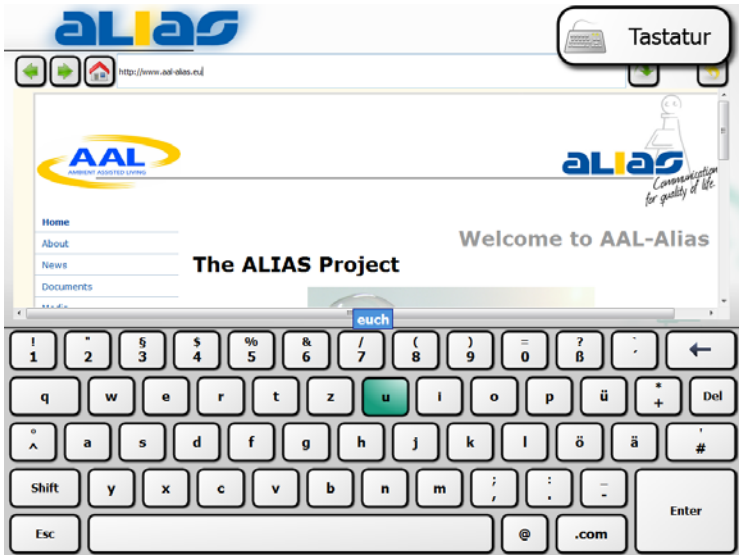


Figure 15: The Hot Virtual Keyboard in action. Its design has been modified to accommodate the robot's GUI. The key "u" has just been pressed. The current input sequence is a web URL with input top-level domain "eu," hence the keyboard's German auto completion "euch" (engl. you) is displayed at the center of the screen.

The Hot Virtual Keyboard uses a special ALIAS design, which matches the design of the robot's GUI. Again, each key has a distinct high contrast frame and indicates its being pressed by a suitable animation. The special ALIAS design prevents the keyboard from being moved or resized by the user – unintentionally or otherwise.

The robot's GUI retains full control over the keyboard, its design and geometry. This is achieved by means of various Windows API functions as they are described in the Hot Virtual Keyboard development center[9]. In short the keyboard is controlled by modifying the Windows registry, altering the keyboard window directly and adjusting the robot's GUI below the keyboard accordingly. This provides the illusion of the virtual keyboard being part of the GUI while maintaining its system-level

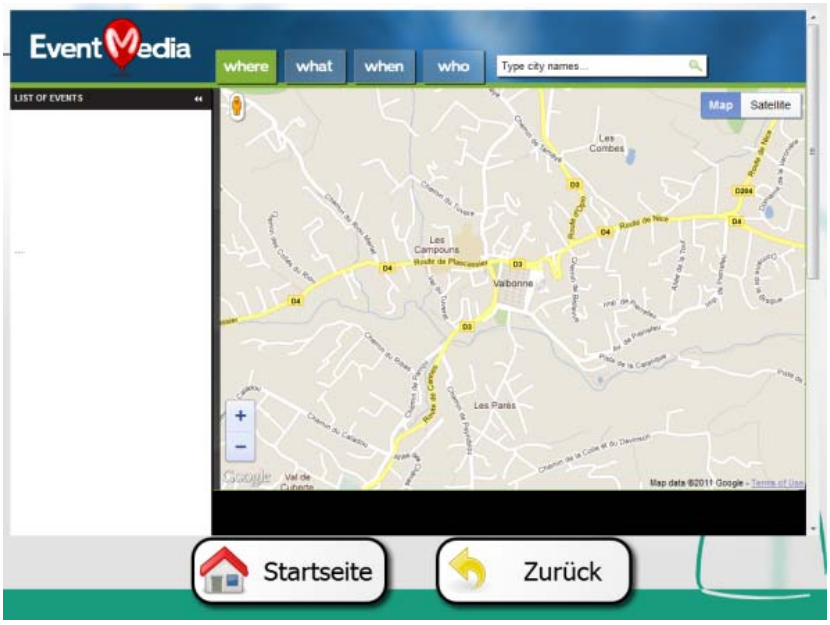
[9] Hot Virtual Keyboard Development Center, http://hot-virtual-keyboard.com/development/

integrity. So the keyboard retains the ability to provide inputs to third-party applications that are not part of the robot's GUI.

As the latest addition to the robot's GUI, the virtual keyboard isn't included in the GUI's interface, yet. So at this point it cannot be controlled by the Dialogue Manager.

3.5. Event Search Engine

The integration of the Event Search Engine is inherently realized by the browser integration, since it runs detached from the actual GUI inside the GUIs web-browser module. But the Search Engine imposes several feature requirements on the web-browser that have to be met. For one the web-browser integration must be stable. In addition it is supposed to support geo-location and other HTML 5 features. Thus, integration of the Event Search Engine using the current web-browser is still subject to current work. The current status is shown in the figure below.



4. Interfaces to Software Modules

This section focuses on the interaction between the Dialogue Manager (DM), the robot's GUI and the other software modules. DM and GUI communicate via network interface using the User Datagram Protocol (UDP) in order to exchange data packages, containing text messages. These packages are composed according to Extensible Markup Language (XML) guidelines, but neglect the explicit declaration of a Document Type Definition (DTD). For character encoding the common 8-bit Universal Character Set Transformation Format (UTF-8) is used, without specification of a byte order marks (BOM). This allows for the messages to contain non-English special characters.

There are three types of packages: "command," "signal," and "request." Commands are issued in order to trigger the GUI to perform a specific task, e.g. switching a menu. Signals are notifications, used to propagate recent status updates, e.g. the successful execution of a previously issued command. Requests are commands, in need for verification. They are sent to the Dialogue Manager to be acknowledged and returned as a command, if the Dialogue Manager deems this to be okay. Table 1 provides a brief overview of common communication patterns between GUI and Dialogue Manager.

Table 1: Typical communication patterns between GUI and Dialogue Manager.

Commands (Sender → Receiver)	Data structure
External commands from the Dialogue Manager (e.g. speech input)	
DM → GUI GUI → DM	command signal
Internal commands from the GUI	
GUI → GUI GUI → DM	command signal
Commands from the GUI via the Dialogue Manager (e.g. starting a program)	
GUI → DM DM → GUI GUI → DM	request (to be returned as a command) command signal
GUI → DM	request (ignored by the Dialogue Manager)
Relevant changes in the GUI or its modules (e.g. incoming phone call)	
GUI → DM	signal

The following text will present the three data package types in closer detail. Package structures will be visualized using the DTD specifications, as they might precede the related package. These DTDs are templates for the related packages, though they're excluded from the actual transmission, saving some bandwidth.

Every menu and every integrated application has a unique identification string that will be used within the data packages in order to address them.

4.1. Command Package

Commands are as the name suggests commands to be executed by the GUI, e.g. switching a menu or starting a game application. Each command package is composed according to the DTD template as presented in Figure 16.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE command [
<!ELEMENT command (arg*)>
<!ATTLIST module CDATA #REQUIRED
          id     CDATA #REQUIRED
          value  CDATA "">
<!ELEMENT arg CDATA>
<!ATTLIST arg
          name  CDATA #REQUIRED>
]>
```

Figure 16: Document type description for the command XML structure. (To be excluded from the actual transmission.)

The *command* attributes *module* and *id* are mandatory for every command. Whereas *module* contains the command category identification string, i.e. the module to which the commands belongs. IDs “menu” and “app” are reserved for switching to a specific menu or for starting an (integrated) application. The *module* attribute is also used for addressing a specific application; in this case *module* is used to store the application’s identification string.

The second mandatory attribute *id* represents the actual command’s identification string whose actual meaning depends on the provided *module* attribute. In case *module* is “menu”, the *id* attribute contains the ID of the menu to be shown on the screen. With *module* set to “app” the *id* attribute contains the identification string of the application to be executed. Otherwise, in case *module* contains an application ID, the meaning of the *id* attribute is up to the specified application.

The third *command* attribute *value* is optional. Its meaning is depending on the provided combination of *module* and *id* attributes. For commands with *module* “menu” or “app”, there is no *value*. (If specified anyway, it’ll be ignored.)

Optional child nodes *arg* may be used to provide the command with additional arguments or data. The *arg* node’s attribute *name* represents the name of the provided argument, while the node body contains the actual argument.

Table 2 provides a short overview of currently available commands. Please note, that the command for closing a started application is implemented by the application itself, not the menu system. So some applications may not support a “closed” command ID. Basically each application module may have its very own set of commands and signals (see below.)

Table 2: Overview of available commands.

Attributes			Meaning
module	*Id*	*value*	
menu	id_mainmenu		Switch to the main menu.
	id_games		Switch to the games menu.
app	id_skype		Run the telephone (Skype) application.
	id_chess		Run the Chess game application.
	id_sudoku		Run the Sudoku game application.
	id_tictactoe		Run the Tic-Tac-Toe game application.
	id_solitaire		Run the Solitaire game application.
	id_webbrowser		Run the web-browser application.
In case the telephone (Skype) application is running, already.			
id_skype	call_by_name	sven.alias	Call Skype-User with ID „sven.alias“.
	call_by_number	+49123456…	Call to phone number *+49123456…* using the public switched telephone network (PSTN.)
	call_disconnect		Disconnect the current phone call.
In case an application has been started. (Every application module features its very own individual functionality, so there is no guarantee that this function will be supported by all future modules.)			
id_skype id_chess id_sudoku id_tictactoe id_solitaire id_webbrowser	close		Close the previously started application and return to the previous menu.

4.2. Signal Package

The signal type packages are used to acknowledge received commands, propagate user inputs, or to provide the Dialogue Manager with GUI status' updates, e.g. an incoming phone call. Signals may be a direct or asynchronous response to a received command, the may be affiliated to a specific module or occur at random, e.g. an incoming phone call. The actual package structure is similar to the *command* package; it is composed according to the DTD template as presented in Figure 17.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE signal [
<!ELEMENT signal (arg*)>
<!ATTLIST module CDATA #REQUIRED
          id     CDATA #REQUIRED
          value  CDATA "">
<!ELEMENT arg CDATA>
<!ATTLIST arg
          name  CDATA #REQUIRED>
]>
```

Figure 17: Document type description for the signal XML structure. (To be excluded from the actual transmission.)

The attributes *module* and *id* are used in the same manner as presented in the command package description, to affiliate the signal to a certain module and command. In most cases the *value* attribute is used to store the current status, which is either “ok” or “error”; however this depends on the provided *module* and *id*. After all the value attribute is optional and may be empty or missing as well.

As with the command package, optional child nodes *arg* may be used to provide the additional arguments or data. The *arg* node’s attribute *name* represents the name of the provided argument, while the node body contains the actual argument. Table 3 contains short overview of possible signals and their purposes.

Table 3: Overview of available signals.

Attributes			Meaning
module	*id*	*value*	
menu	id_mainmenu id_games	ok error	Switching to the requested menu has been initialed (“ok”) or it has failed (“error”.)
app	id_skype id_chess id_sudoku id_tictactoe id_solitaire id_webbrowser	ok error	The requested application is going to be started (“ok”) or it won’t (“error”.)
id_skype id_chess id_sudoku id_tictactoe id_solitaire id_webbrowser	start	ok error	The application has been started (“ok”) or it failed (“error”.)
id_skype id_chess id_sudoku id_tictactoe id_solitaire id_webbrowser	close	ok error	The application has been closed (“ok”) or it hasn’t (“error”.)
In case the telephone (Skype) application is running, already.			
id_skype	call_by_number call_by_name call_disconnect	ok error	The requested command has been executed (“ok”) or it failed (“error”)

4.3. Request Package

The *request* package is exactly the same as the *command*, with one exception, the root node is named *request* instead of *command*. In case the GUI needs to execute a command itself, it generates a request package and sends it to the Dialogue Manager. Doing so enables the Dialogue Manager to

asses the requested and issues an appropriate command to the GUI, or ignores the request if deemed inappropriate. The request package's DTD is presented in Figure 18.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE request [
<!ELEMENT request (arg*)>
<!ATTLIST module CDATA #REQUIRED
          id     CDATA #REQUIRED
          value  CDATA "">
<!ELEMENT arg CDATA>
<!ATTLIST arg
          name  CDATA #REQUIRED>
]>
```

Figure 18: Document type description for the request XML structure. (To be excluded from the actual transmission.)

4.4. Example

A short exemplary communication between Dialogue Manger module and the robot's GUI is presented in Figure 19; it demonstrates the initiation of a phone call using the public switched telephone network (PSTN.)

```
DM → GUI     <command module="app" id="id_skype"/>
GUI → DM     <signal module="app" id="id_skype" value="ok"/>
GUI → DM     <signal module="id_skype" id="start" value="ok"/>
DM → GUI     <command module="id_skype" id="call_by_number" value="+49123..."/>
GUI → DM     <signal module="id_skype" id="call_by_number" value="ok"/>

... phone conversation ...

GUI → GUI    <command module="id_skype" id="call_disconnect"/>
GUI → DM     <signal module="id_skype" id="call_disconnect" value="ok"/>
DM → GUI     <command module="id_skype" id="close"/>
GUI → DM     <signal module="id_skype" id="close" value="ok"/>
DM → GUI     <command module="menu" id="id_mainmenu"/>
GUI → DM     <signal module="menu" id="id_mainmenu" value="ok"/>
```

Figure 19: Exemplary communication between the Dialogue Manger (DM) module and the robot's GUI. The DM starts the Skype module and initiates a phone call to the specified phone number. The call is disconnected via the GUI thereafter the DM decides to switch the GUI back to the main menu.

Note that the example as presented in Figure 19 contains a command that is sent by the GUI to the GUI itself. Alternatively the GUI could have sent a request for the desired command to the Dialogue Manager which in turn could have provided it with the requested command. Or the Dialogue Manager could discard the GUI's request, for various reasons. In this example there is no point in preventing the user from disconnecting the call via the GUI, so the GUI doesn't bother the Dialogue Manager with a request. Instead the Dialogue Manager will be kept informed by the GUI's status signals.

4.5. Planed Extensions

With the planned addition of a contacts database the interface will be further extended to offer access to the Skype contact list and their online statuses. The web-browser module is to be accommodated by some basic navigation commands. Additional modules like an audio book section or user inputs via the brain computer interface (BCI) need to be accounted for, also.

5. Summary and Conclusions

Integrating external applications into a common framework and definition of the respective interfaces was described in this deliverable. Some applications like Skype offer special APIs for integration. Though these APIs are not always compatible, they pave the way for proper integration.

Third-party programs without a special integration API of their own may be integrated by using native Windows API functions. This is a very tricky process. Several different approaches have been tried with varying success. Generally, there is no proper one-fits-all solution and every single application requires special attention.

Software updates of integrated applications may cause problems because sometimes they completely invalidate previous integration attempts. This is especially hazardous since most nowadays-applications include automatic update features. Due to e.g. the more frequent update policy of the Firefox web-browser it became unsuitable for integration to the ALIAS GUI. So the current premise is to use the QtWebKit browser engine instead.

Depending on the individual application and level of integration, there are different functions that can be controlled by the robot's GUI. Some of these functions have been added to the GUI's interface to the Dialogue Manager. So the Dialogue Manager is able to remote control relevant functions of the GUI and place a phone call for example.

The interfaces are composed of XML structured data packages, sent via UDP network protocol. This system will need to be extended to account for the planed contacts database, that'll be implemented by the Dialogue Manager. The web-browser application and the upcoming new version of the Skype module need to be fully incorporated to the interface. In addition the GUI needs to be adapted for the integration of the robot's brain interface.