

IST Amigo Project Deliverable D4.7

Intelligent User Services

5 - User Interface Service Software Developer's Guide

IST-2004-004182

Public



| | | |
|-------------------------|---|------------|
| Project Number | : | IST-004182 |
| Project Title | : | Amigo |
| Deliverable Type | : | Report |

| | | |
|----------------------------------|---|---|
| Deliverable Number | : | D4.7 (UIS contribution) |
| Title of Deliverable | : | 5 - UIS Software developer's guide |
| Nature of Deliverable | : | Public |
| Internal Document Number | : | amigo_5_d4.7_final |
| Contractual Delivery Date | : | 30 November 2007 |
| Actual Delivery Date | : | 14 January 2008 |
| Contributing WPs | : | WP4 |
| Author(s) | : | Basilis Kladis (SinL), Christophe Cerisara (INRIA), Sinisa Dukanovic (Fraunhofer Institute SIT), Edwin Naroska (Fraunhofer Institute IMS) |

Abstract

This document is the final programmers guide for the User Interface Service components. It describes how to install, use and edit the components of UIS.

Keyword list

Speech interface, speech recognition, speech synthesis, speaker recognition, 3D gesture service, 2D gesture, gesture service, multimodal dialogue manager, multimodal fusion, implicit speech input, MMIL, GUI Service.

Table of Contents

| | |
|---|-----------|
| Table of Contents..... | 2 |
| 1 Speech based Interface | 6 |
| 1.1 Component Overview | 6 |
| 1.1.1 Voice Service | 6 |
| 1.1.1.1 Speaker Recognition Component | 6 |
| 1.1.1.2 Explicit Speech Interaction..... | 8 |
| 1.2 Deployment | 10 |
| 1.2.1 Voice Service | 10 |
| 1.2.1.1 Speaker Recognition Component | 10 |
| 1.2.1.2 Explicit Speech Interaction | 10 |
| 1.3 Component Architecture | 21 |
| 1.3.1 Speaker recognition component interface | 21 |
| 1.3.1.1 Capability ISpeakerRecognition::createUser() | 21 |
| 1.3.1.2 Capability ISpeakerRecognition::queryUser() | 21 |
| 1.3.1.3 Capability ISpeakerRecognition::clearUser() | 21 |
| 1.3.1.4 Capability ISpeakerRecognition::deleteUser() | 22 |
| 1.3.1.5 Capability ISpeakerRecognition::enrollUser()..... | 22 |
| 1.3.1.6 Capability ISpeakerRecognition::verifyUser() | 23 |
| 1.3.1.7 Capability ISpeakerRecognition::identifyUser()..... | 23 |
| 1.3.2 Explicit speech interaction component interface..... | 24 |
| 1.3.2.1 Capability IVoiceIO::ListenToUser() | 24 |
| 1.3.2.2 Capability IVoiceIO::ListenToUserBL() | 26 |
| 1.3.2.3 Capability IVoiceIO::interact()..... | 27 |
| 1.3.2.4 Capability IVoiceIO::interactBL() | 27 |
| 1.3.2.5 Capability IVoiceIO::interactEH() | 27 |
| 1.3.2.6 Capability IVoiceIO::SpeakToUser()..... | 28 |
| 1.3.2.7 Capability IVoiceIO::SpeakToUserAdv()..... | 29 |
| 1.3.2.8 Capability IVoiceIO::GenerateResponse()..... | 29 |
| 1.4 Tutorial..... | 31 |
| 1.4.1 SpeakerRecognition Service..... | 31 |
| 1.4.1.1 Install SpeakerRecognition | 31 |
| 1.4.1.2 Run SpeakerRecognition Server..... | 35 |
| 1.4.1.3 Run Trainer Demo Application | 36 |
| 1.4.1.4 Run Identifier Demo Application | 38 |
| 1.4.1.5 Run Verifier Demo Application | 41 |
| 1.4.2 Explicit Speech Interaction Service..... | 43 |
| 1.4.2.1 Install ASR and TTS engines | 43 |
| 1.4.2.2 Configure ASR and TTS engines | 43 |
| 1.4.2.3 Install \ Update ASR and TTS license..... | 43 |
| 1.4.2.4 Install VoiceIO..... | 47 |
| 1.4.2.5 Run VoiceIO Server and Client Application..... | 49 |
| 1.4.2.6 Run VoiceIO Server and GenSynthClient Application | 51 |
| 2 3D Gesture Service..... | 54 |

| | |
|---|-----------|
| 2.1 Component Overview | 54 |
| 2.1.1 Gesture Service..... | 54 |
| 2.1.1.1 3D Gesture Service | 54 |
| Provider | 54 |
| 2.2 Deployment | 56 |
| 2.2.1 Gesture Service..... | 56 |
| 2.2.1.1 3D Gesture Service | 56 |
| 2.3 Component Architecture | 58 |
| 2.3.1 Gesture Service..... | 58 |
| 2.3.1.1 3D Gesture service i..... | 58 |
| 2.4 Tutorial..... | 60 |
| 2.4.1 3D gesture Service..... | 60 |
| 2.5 Deployment | 62 |
| 2.5.1 Gesture Service..... | 62 |
| 2.5.1.1 3D Gesture Service | 62 |
| 2.6 Tutorial..... | 64 |
| 2.6.1 3D gesture Service..... | 64 |
| 2.6.1.1 Component development | 64 |
| 2.6.1.2 Application development..... | 64 |
| 3 Multi-modal Interface Services / Multi-device and Dialog Management Service..... | 68 |
| 3.1 Component Overview | 68 |
| 3.1.1 Voice Service | 68 |
| 3.1.1.1 Implicit Speech Input..... | 68 |
| 3.1.2 Gesture Service..... | 70 |
| 3.1.2.1 2D Gesture Service | 70 |
| 3.1.3 Dialogue Manager | 71 |
| 3.1.4 Multimodal Fusion | 73 |
| 3.2 Deployment | 75 |
| 3.2.1 Implicit Speech Input | 75 |
| 3.2.1.1 System requirements..... | 75 |
| 3.2.1.2 Download..... | 75 |
| 3.2.1.3 Install..... | 75 |
| 3.2.1.4 Configure | 75 |
| 3.2.1.5 Compile..... | 75 |
| 3.2.2 2D Gesture Service..... | 75 |
| 3.2.2.1 System requirements..... | 75 |
| 3.2.2.2 Download..... | 76 |
| 3.2.2.3 Install..... | 76 |
| 3.2.2.4 Configure | 76 |
| 3.2.2.5 Compile..... | 79 |
| 3.2.3 Multimodal Dialogue Manager | 79 |
| 3.2.3.1 System requirements..... | 79 |
| 3.2.3.2 Download..... | 79 |
| 3.2.3.3 Install..... | 79 |

| | | |
|------------|--|------------|
| 3.2.3.4 | Configure | 79 |
| 3.2.3.5 | Compile..... | 79 |
| 3.2.4 | Multimodal Fusion | 79 |
| 3.2.4.1 | System requirements..... | 79 |
| 3.2.4.2 | Download | 80 |
| 3.2.4.3 | Install..... | 80 |
| 3.2.4.4 | Configure | 80 |
| 3.2.4.5 | Compile..... | 80 |
| 3.3 | Component Architecture | 81 |
| 3.3.1 | Implicit speech input component interface..... | 81 |
| 3.3.2 | Gesture Service..... | 82 |
| 3.3.2.1 | 2D Gesture service interfaces | 82 |
| 3.3.2.2 | GestureInterpreterService | 82 |
| 3.3.3 | Multimodal Dialogue Manager | 83 |
| 3.3.3.1 | Limitations and scope of services..... | 83 |
| 3.3.3.2 | Integration with the Context Management Service..... | 84 |
| 3.3.3.3 | Dialogue Manager Internals | 85 |
| 3.3.3.4 | Component interfaces | 87 |
| 3.3.4 | Multimodal Fusion | 87 |
| 3.3.4.1 | Component interface..... | 87 |
| 3.4 | Tutorial..... | 89 |
| 3.4.1 | Gesture Service..... | 89 |
| 3.4.1.1 | 2D Gesture Service | 89 |
| 3.4.2 | Multimodal Fusion component development | 91 |
| 3.4.3 | Multimodal Fusion application development | 91 |
| 3.4.4 | Dialogue Manager | 94 |
| 3.5 | Assessment | 95 |
| 3.5.1 | Voice Service | 95 |
| 3.5.1.1 | Implicit Speech Input..... | 95 |
| 3.5.2 | Dialogue Manager | 95 |
| 3.5.3 | 2D Gesture and Multimodal Fusion | 95 |
| 3.6 | Appendix | 97 |
| 3.6.1 | Multimodal Fusion sample files and messages | 97 |
| 3.6.1.1 | Properties file for time windows: | 97 |
| 3.6.1.2 | MMIL message example for gesture | 97 |
| 3.6.1.3 | MMIL message example for selection | 97 |
| 3.6.1.4 | MMIL message example for sentence..... | 97 |
| 3.7 | GUI-Service Component Overview | 98 |
| 3.8 | Deployment GUI Service | 100 |
| 3.8.1 | System requirements | 100 |
| 3.8.2 | Download | 100 |
| 3.8.3 | Install..... | 100 |
| 3.8.4 | Configure..... | 100 |
| 3.9 | Component Architecture GUI Service | 101 |
| 3.9.1 | Interfaces | 102 |
| 3.9.2 | Classes of the UI-Service | 103 |

| | | |
|-------------|--|------------|
| 3.9.3 | Classes used for navigation | 104 |
| 3.10 | Assessment GUI Service | 106 |
| 3.10.1 | User Acceptance..... | 106 |
| 3.10.1.1 | Study Methodology and Hypothesis | 106 |
| 3.10.1.2 | Test apparatus | 106 |
| 3.10.2 | Test Results | 110 |
| 3.10.3 | System Performance..... | 113 |
| 4 | Appendix | 114 |

1 Speech based Interface

1.1 Component Overview

The overall functional architecture of the Speech based interface has been described in deliverable D4.1 and D4.2. The Voice interface service consists of the following main components: Explicit Speech Interaction service, Speaker Recognition Service and Implicit Speech Input Service.

The introduced main components and their subcomponents are further treated in this document.

1.1.1 Voice Service

1.1.1.1 Speaker Recognition Component

Provider

SingularLogic S.A.

Introduction

Speaker recognition component provides two basic functionalities: enrollment of new users and recognition of enrolled users.

- *Enrollment* happens when a new user joins the system – a person speaks and the application builds a *voiceprint model* for that person.
- *Recognition* happens in the future whenever the application wants to ensure the identity of a speaker – the user speaks and the speech is compared to existing voiceprint models. If the comparison scores above a predefined threshold, the identity is validated; otherwise the identity claim is rejected.

Speaker recognition component includes: speaker recognition engine and speaker recognition server.

Development status

Speaker recognition engine software is available as WIN32 dynamic linked library (DLL).

Speaker recognition server is deployed as an autonomous executable in .NET framework. Final versions available at INRIA Gforge repository.

Intended audience

Project partners

License

Speaker recognition engine:

SingularLogic S.A. owns the IPR for speaker recognition engine software. Licensing negotiable with separate agreement.

Speaker recognition server.

LGPL

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Hardware:

- PC/Laptop computer with network connection and microphone.

Software:

- Windows 2K
- .NET 2.0
- Amigo discovery framework

Platform

Microsoft .NET Framework v2.0

Tools

Microsoft Visual Studio 2005

Files

Speaker recognition engine:

Time-limited speaker recognition engine library (dll) available at SingularLogic S.A.

Speaker recognition server.

Final version available for download at:

[amigo_gforge]/ius/user_interface/voice_service/speaker_recognition.

Documents

Component design and architecture can be found in D4.1 and D4.2. Software developer's guide and installation guide are provided in this document.

Tasks

-

Bugs

-

Patches

-

1.1.1.2 Explicit Speech Interaction**Provider**

SingularLogic S.A.

Introduction

Explicit speech interaction provides means for user-system communication using natural language dialogues. Two layers of abstraction can be distinguished:

- *Explicit speech interaction framework*, that includes general-purpose modules responsible for start-up, shutdown, initialization and communication tasks of speech interface as well as speech application execution environment, and,
- *Speech applications resources*, that are task-oriented collections of resources and scripts able to provide a complete interaction for fulfilling the specific tasks.

Development status

Final version for .NET available.

Intended audience

Project partners

License

LGPL

- recognition and synthesis engines are commercial software covered by vendors' licenses. SingularLogic provides a project-life free license to the consortium.

Language

C#, GRXML, SSML

Environment (set-up) info needed if you want to run this sw (service)

Hardware:

- PC/Laptop computer with network connection, microphone, speakers

Software:

- Windows 2000 SP4

- Speech recognition engine Nuance OSR3.0
- TTS Nuance RealSpeak 4.0.1

Platform

Microsoft .NET Framework v2.0

Tools

Microsoft Visual Studio 2005

Files

Current version of explicit speech interaction framework available for download at:

- [amigo_gforge]/ius/user_interface/voice_service/explicit_speech
- Nuance OSR3 speech recognition engine available for download at: www.nuance.com
- Nuance RealSpeak TTS engine available for download at: www.nuance.com
- Language packs for recognition and synthesis are provided by SingularLogic S.A.

Documents

Component design and architecture can be found in D4.1 and D4.2. Software developer's guide and installation guide are provided in this document.

Tasks

-

Bugs

-

Patches

-

1.2 Deployment

1.2.1 Voice Service

1.2.1.1 Speaker Recognition Component

1.2.1.1.1 System requirements

Amigo .NET programming framework

Microsoft .NET 2.0 Framework v2.0

1.2.1.1.2 Download

The run time executables are included in "SpeakerRecognition.msi", located at:

[amigo]/User Interface/

1.2.1.1.3 Install

The installation package will install all required software and configuration files (default destination C:\Amigo). For a detailed step-by-step installation refer to SpeakerRecognition tutorial.

1.2.1.1.4 Configure

The SpeakerRecognition engine SpkRec.dll should be registered before use. From command line enter the following command:

```
>regsvr32 SpkRec.dll
```

New speakers should be enrolled before using the Speaker Identification or Speaker Verification capabilities. Refer to SpeakerRecognition tutorial for a step-by-step training process.

1.2.1.1.5 Compile

Source code is available as C# solution in SpeakerRecognition-src.zip located at:

[amigo]/User Interface/

Use Microsoft Visual Studio 2005 to compile a new version of SpeakerRecognition Service.

1.2.1.2 Explicit Speech Interaction

1.2.1.2.1 System requirements

Amigo .NET programming framework

Microsoft .NET 2.0 Framework v2.0

1.2.1.2.2 Download

The run time executables are included in "VoicelOServer.msi", located at:
[amigo]/User Interface/

1.2.1.2.3 Install

The installation package will install all required software and configuration files (default destination C:\Amigo)

Note: OSR3.0 recognition engine and RealSpeak 4.0.1 synthesis engine as well as the desired language packs should be installed in order to be able to use VoicelO server.

1.2.1.2.4 Configure

Configuration of speech recognition engine, TTS engine and VoicelO server are required.

Configuration settings of OSR3.0 ->Files: SpeechWorks.cfg, user.xml

Configuration guide is included in the installation package of the engine. Sample configuration files are provided here:

SpeechWorks.cfg

```
REGEDIT4
[HKEY_LOCAL_MACHINE]
[HKEY_LOCAL_MACHINE\SOFTWARE]
[HKEY_LOCAL_MACHINE\SOFTWARE\SpeechWorks International]
[HKEY_LOCAL_MACHINE\SOFTWARE\SpeechWorks International\OpenSpeech Recognizer]
[HKEY_LOCAL_MACHINE\SOFTWARE\SpeechWorks International\OpenSpeech Recognizer\3.0]
DataCaptureDirectory=$SWISRSDK\data
DataCaptureDiskSpace=1000000
DiagConfigPollSecs=600
DiagErrorMap=$SWISRSDK/config/SWIErrors.en.us.txt
DiagFileName=$SWISRSDK/trc.log
DiagMaxFileSizeKB=1024
DiagOutputStreamTypes=debug,file
DiagSuppressTimestamp=
DiagTagMapsBaseline=$SWISRSDK/config/OSRServerTagMap.txt;$SWISRSDK/config/defaultTagmap.txt;$SWISRSDK/config/bwcompatTagmap.txt
DiagTagMapsPlatform=
DiagTagMapsUser=
SWILicenseServerList=27000@localhost
SWIUSERCFG=$SWISRSDK/config/user.xml
SWIsvcDiagFileName=$SWISRSDK/config/svctr.log
SWIsvcMonAutoRestart=
SWIsvcMonAutoStart=
SWIsvcMonDiagFileName=$SWISRSDK/config/montrc.log
SWIsvcMonDiagMaxFileSizeKB=
```

```
SWIsvcMonPort=
SWIsvcServerExecutiveProcess=$SWISRSdk/bin/OSRServer.exe
DefaultLanguage=default
```

user.xml

```
<?xml version="1.0"?>      <!-- XML version -->
<!-- ***** CONFIG PARAMETERS ***** -->
<SWIrecConfig version="1.0.0">
  <lang name="default">
    <param name="swirec_audio_media_type">
      <value>audio/basic;rate=8000</value>
      <value>audio/L 16;rate=8000</value>
    </param>
    <param name="swiep_audio_media_type">
      <value>audio/basic;rate=8000</value>
      <value>audio/L 16;rate=8000</value>
    </param>
    <param name="swiep_mode">
      <value>begin_end</value>
    </param>
    <param name="timeout">
      <value>7000</value>
    </param>
    <param name="incompletetimeout">
      <value>1500</value>
    </param>
  </lang>
</SWIrecConfig>
```

Configuration settings of RealSpeak4.0 ->Files: swittsclient.cfg, ttsserver.xml

Configuration guide is included in the installation package of the engine. Sample configuration files are provided here:

swittsclient.cfg

```
[Header]
# Copyright (c) 2004 ScanSoft, Inc. All rights reserved.
#
# This file is used to maintain Speechify 2.1/3.0 and Speechify Solo 1.0
# compatibility for the SWItts API.
[DefaultDictionaries]
# Speechify 2.1, 3.0, and Speechify Solo 1.0 supported default
# dictionaries that were automatically loaded for each port when the
```

```

# port was created. It looked for language specific and voice specific
# files with very specific names, where there could be one language
# specific dictionary and/or one voice specific dictionary for each of
# the Speechify dictionary types for that language:
#
# <IETF lang ID>/<IETF lang ID>-<dict type>.dic
# <IETF lang ID>/<voice name>/<voice name>-<dict type>.dic
#
# This section of this configuration file permits obtaining that
# functionality in a more flexible way.
#
# FORMAT:
#
# Each line in this section defines a RealSpeak dictionary to load and
# activate when a port is created (Speechify dictionaries must be
# converted to RealSpeak dictionaries using an offline migration tool,
# see the Speechify to RealSpeak migration guide for details). Lines
# starting with a # are comments. Each non-comment line has the
# following format, where <voice> can be "*" to specify all voices:
#
# <language> <voice> <dict priority> <dict URI> <dict content type>
#
# For example, the following line specifies that when a port is opened
# for the en-US Tom voice, en-US.dct and Tom.dct are loaded and activated
# where Tom.dct is a text format dictionary, en-US.dct is a binary format
# dictionary, and Tom.dct takes precedence. If en-US Jennifer is
# loaded instead, only en-US.dct will be loaded and activated.
#
# en-US * 1 /dictionaries/en-US.dct application/edct-bin-dictionary
# en-US Tom 2 /dictionaries/Tom.dct application/edct-text-dictionary
#
[PortMap]
# In the SWItts API for Speechify 2.1 and 3.0, a host name/host port
# pair are used to specify a Speechify server. Since each Speechify
# server instance only supports a single language/voice, the port
# number was also sufficient to indicate the desired
# language/voice. However, in the RealSpeak environment, a single
# RealSpeak server instance can support many languages/voices
# simultaneously.
#
# To support smooth migration to RealSpeak server environments, this
# file defines mappings from Speechify port numbers to the actual

```

```
# RealSpeak port, language, and voice that will actually be
# used. Since Speechify 2.1/3.0 permitted customizing the port numbers
# for their environment, customers may need to update this table to
# account for their own particular configuration.
#
# FORMAT:
#
# Each line in this section defines a Speechify 2.1/3.0 -> RealSpeak
# server mapping. Lines starting with a # are comments. Each
# non-comment line has the following format:
#
# <Speechify port> <RS port> <language> <voice> <frequency>
#
# For example, the following line specifies that when
# SWltsOpenPort(ttsPort, "myhost", 5573, MyCallback, NULL) is called,
# a connection should be established to myhost:6666 and the en-US tom
# 8 kHz voice is used.
#
# 5573 6666 en-US tom 8000
#
5555 6666 en-US Mara 8000
5557 6666 en-US Mara 11000
5556 6666 en-US Tom 8000    # Tom replaces Speechify 2.1 Rick
5558 6666 en-US Tom 11000
5559 6666 fr-FR Sophie 8000 # Sophie replaces Speechify 2.1/3.0 Sandra
5560 6666 fr-FR Sophie 11000
5561 6666 es-MX Javier 8000 # Javier replaces Speechify 2.1/3.0 Paulina
5562 6666 es-MX Javier 11000
5563 6666 en-GB Helen 8000
5564 6666 en-GB Helen 11000
5565 6666 de-DE Tessa 8000
5566 6666 de-DE Tessa 11000
5567 6666 ja-JP Kyoko 8000  # Kyoko replaces Speechify 2.1/3.0 Kimiko
5568 6666 ja-JP Kyoko 11000
5569 6666 es-MX Javier 8000
5570 6666 es-MX Javier 11000
5571 6666 pt-BR Luci 8000
5572 6666 pt-BR Luci 11000
5573 6666 en-US Tom 8000
5574 6666 en-US Tom 11000
5575 6666 es-ES Marta 8000
```

5576 6666 es-ES Marta 11000
 5577 6666 en-AU Karen 8000
 5578 6666 en-AU Karen 11000
 5579 6666 fr-CA Felix 8000
 5580 6666 fr-CA Felix 11000
 5581 6666 en-US Jill 8000
 5582 6666 en-US Jill 11000
 5583 6666 ja-JP Kyoko 8000
 5584 6666 ja-JP Kyoko 11000

ttsserver.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?> <!-- XML version -->
<?xml-stylesheet type="text/xsl" href="ttsserver.xsl"?>
<!--
RealSpeak Host 4.0 Configuration File
*****License*****
Copyright © 1995-2004 by ScanSoft, Inc. All rights reserved.
ScanSoft, Inc. provides this document without representation or warranty of
any kind. ScanSoft, Inc. reserves the right to revise this document and to
change the information contained in this document without further notice.
RealSpeak, Dialog Modules, OpenSpeech, Productivity Without
Boundaries, ScanSoft, the ScanSoft logo, SMARTRecognizer, SpeechCare,
Speechify, SpeechSecure, SpeechSpot, SpeechSite, SpeechWorks, the
SpeechWorks logo, and SpeechWorksHere are trademarks or registered
trademarks of ScanSoft, Inc. or its licensors in the United States
and/or other countries. This document may also contain other
trademarks, which are the property of their respective owners.
Without limiting the rights under copyright reserved above, no part of
this document may be reproduced, stored in or introduced into a
retrieval system, or transmitted in any form or by any means,
including, without limitation, electronic, mechanical, photocopying,
recording, or otherwise, without the prior written permission of
ScanSoft, Inc.
-->
<ttsserver version="4.0.0" xmlns="http://www.scansoft.com/rsh40/ttsserver">
  <!-- *** ENVIRONMENT VARIABLE OVERRIDES *** -->
  <!-- Installation directory, by default auto-detected -->
  <!-- <SSFTTTSSDK></SSFTTTSSDK> -->
  <!-- Temporary files directory, used below, by default auto-detected -->
  <!-- <TMPDIR></TMPDIR> -->
  <!-- User ID, used below, by default auto-detected -->
  <!-- <USER></USER> -->
  <!-- *** NETWORK PARAMETERS *** -->
```



```

<!-- TCP/IP service name, if empty network_port is used -->
<network_service></network_service>
<!-- TCP/IP port number -->
<network_port>6666</network_port>
<!-- TCP/IP backlog for accepting connections -->
<network_accept_backlog>20</network_accept_backlog>
<!-- Maximum number of connections before refusing clients -->
<network_client_limit>1000</network_client_limit>
<!-- Whether to allow the server to listen for connections on
an already active TCP/IP port. By default, false, as doing
so exposes a well-known security flaw where other
processes could hijack the port afterwards. Only switch the
value to true if directed by ScanSoft Technical Support to
workaround OS problems with releasing the TCP/IP port on
shutdown. -->
<network_reuse_addr>false</network_reuse_addr>
<!-- Network interfaces to listen for connections upon. By default,
the server listens for connections on all network interfaces.
Uncomment and set this to enhance security in cases where
the server should only accept clients from the same host (use a
value of 127.0.0.1), or where the server should only accept
clients from a single trusted network interface (use the TCP/IP
address for that interface). -->
<!-- <network_interface>127.0.0.1</network_interface> -->
<!-- Network timeout, in seconds, for the client/server
connection. The worst-case for detecting a lost connection
is 2 times this value. -->
<network_timeout>60</network_timeout>
<!-- *** LICENSE PARAMETERS *** -->

<!-- Licensing mode, default or explicit -->
<license_mode>default</license_mode>
<!-- License servers, must have one or more -->
<license_servers>
  <server>27000@localhost</server>
  <!-- <server>27000@myserver.mycompany.com</server> -->
</license_servers>
<!-- *** MISCELLANEOUS PARAMETERS *** -->
<!-- Default path for user dictionaries -->
<dictionary_default_path></dictionary_default_path>
<!-- Run in the background versus as an interactive process -->
<run_in_background>false</run_in_background>

```

```

<!-- Whether to produce core files on crashes on UNIX variants -->
<produce_core_files>true</produce_core_files>
<!-- *** INTERNET FETCH CACHE PARAMETERS *** -->
<!-- Directory name for the disk cache. If relative, the file path
      is relative to the containing configuration file. -->
<cache_directory>${TMPDIR}/ttsserver_cache_${USER}</cache_directory>
<!-- Maximum size of the disk cache in MB -->
<cache_total_size>200</cache_total_size>
<!-- Maximum size of a single disk cache entry in MB -->
<cache_entry_max_size>20</cache_entry_max_size>
<!-- Time when an unused disk cache entry gets purged, in seconds -->
<cache_entry_exp_time>3600</cache_entry_exp_time>
<!-- When maximum cache size is reached, what must the cache size be
      reduced to to stop expiring grammars, in MB -->
<cache_low_watermark>180</cache_low_watermark>
<!-- Reserved for future use, leave unchanged -->
<cache_unlock_entries_at_startup>true</cache_unlock_entries_at_startup>
<!-- *** INTERNET FETCH PARAMETERS *** -->
<!-- Address a http proxy server to use, e.g. 127.0.0.1, by default
      no proxy is used (empty value) -->
<inet_proxy_server></inet_proxy_server>
<!-- Port of the http proxy server to use, e.g. 8080, ignored
      unless inet_proxy_server is non-empty -->
<inet_proxy_server_port>8080</inet_proxy_server_port>
<!-- User agent name in HTTP/1.1 headers -->
<inet_user_agent> RealSpeak Host/4.0 </inet_user_agent>
<!-- Whether to accept HTTP cookies -->
<inet_accept_cookies>true</inet_accept_cookies>
<!-- *** INTERNET FETCH EXTENSION MAPPING RULES *** -->
<inet_extension_rules>
  <extension name=".alaw">audio/x-alaw-basic</extension>
  <extension name=".ulaw">audio/basic</extension>
  <extension name=".wav">audio/x-wav</extension>
  <extension name=".L16">audio/L16;rate=8000</extension>
  <extension name=".txt">text/plain</extension>
  <extension name=".xml">text/xml</extension>
  <extension name=".ssml">application/synthesis+ssml</extension>
  <extension name=".bcd">application/edct-bin-dictionary</extension>
  <extension name=".dct">application/edct-text-dictionary</extension>
  <extension name=".tdc">application/edct-text-dictionary</extension>
</inet_extension_rules>
<!-- *** DIAGNOSTIC LOGGING *** -->

```

```

<!-- Whether to log errors and diagnostics -->
<log_file_enabled>true</log_file_enabled>
<!-- Error and diagnostic log file base name. This will have "1.xml"
and "2.xml" appended for the initial and roll-over log file
names. If relative, the file path is relative to the containing
configuration file. If empty, messages will go to standard
output. -->
<log_file_base_name>C:/Temp/ttsserver_log_${USER}_</log_file_base_name>
<!-- Log file maximum size, in MB -->
<log_file_max_size>50</log_file_max_size>
<!-- Diagnostic log level, by default 0, where 0 enables errors,
1 enables errors and warnings, and higher levels enable diagnostic
messages (which may greatly impact performance). -->
<log_level>3</log_level>
<!-- *** DEFAULT DICTIONARIES *** -->
<!-- List of dictionaries to load, where each matching dictionary is
loaded when each port is opened. The language and priority
attributes are required, the voice attribute is optional (if
not specified, the dictionary is loaded for all voices for that
language). The value is the dictionary path or URI. -->
<!--
<default_dictionaries>
<dictionary language="American English" priority="1000">
http://myserver/american_english.bdc
</dictionary>
<dictionary language="American English" voice="Jill" priority="1001">
http://myserver/jill.bdc
</dictionary>
</default_dictionaries>
-->
</ttsserver>

```

Configuration settings of VoicelO server -> VoicelO.cfg, WavelO.cfg

General settings for the recognition and synthesis processes of VoicelO server are stored in file C:\Amigo\VoicelO\VoicelO.cfg.

| Section | Parameter | Description |
|-----------------|--------------------|--|
| VoicelO service | portNumber | The port where the VoicelO Amigo service will be published to |
| | defaultLanguage | The default recognition and synthesis language. It will be used if userAdaptation is set to no, or the user does not exist in UMPS service. |
| | userAdaptation | If set to yes to interface will try to adapt to specific user preferences concerning the language and speech interaction settings. If set to no the default values will be used. |
| Recognition | recordingFolder | The folder where the recognition results are stored for log purposes |
| Synthesis | responseTextFolder | The folder where response texts are stored |

| | | |
|-------------------|--------------------------|---|
| | <i>responsePCMFolder</i> | <i>The folder where response synthesized sound files are stored</i> |
| <i>Generation</i> | <i>templateFolder</i> | <i>The folder that contains the generation templates</i> |
| <i>Recording</i> | <i>ServerIP</i> | <i>The IP of the PC performing the speech recording</i> |
| | <i>ServerPort</i> | <i>The port of the PC performing the speech recording that the streaming is directed to</i> |

The installation package contains a ready-to-use configuration file with default values:

VoiceIO.cfg

[VoiceIO service]

portNumber = 9876

defaultLanguage = en-US

userAdaptation = yes

[Recognition]

recordingFolder = C:\Amigo\SWs

[Synthesis]

responseTextFolder = C:\Amigo\SWs

responsePCMFolder = C:\Amigo\SWs

[Generation]

templateFolder = C:\Amigo\SWs

[Recording]

ServerIP = 127.0.0.1

ServerPort = 9999

General settings for speech recording, playing and basic sound file transformation processes of WaveIO server are stored in file C:\Amigo\lib\WaveIO.cfg.

| Section | Parameter | Description |
|----------------------|------------------------|---|
| <i>Recording_Mic</i> | <i>recordingFolder</i> | <i>The folder where the recordings are store</i> |
| <i>Recording_TCP</i> | <i>ListenerIP</i> | <i>The IP of the PC performing the speech recording</i> |
| | <i>ListenerPort</i> | <i>The port of the PC performing the speech recording that the streaming is directed to</i> |
| <i>Playing_TCP</i> | <i>ListenerIP</i> | <i>The IP of the PC performing the speech playing</i> |
| | <i>ListenerPort</i> | <i>The port of the PC performing the speech playing that the streaming is directed to</i> |

The installation package contains a ready-to-use configuration file with default values:

WaveIO.cfg

[Recording_Mic]

recordingFolder = C:\Amigo\SWs

```
[Recording_TCP]
ListenerIP = 127.0.0.1
ListenerPort = 9999

[Playing_TCP]
ListenerIP = 127.0.0.1
ListenerPort = 8888
```

1.2.1.2.5 Compile

Source code is available as C# solution in VoicelOServer-src.zip located at:

[amigo]/User Interface/

Use Microsoft Visual Studio 2005 to compile a new version of Explicit Speech Interaction Service.

1.3 Component Architecture

1.3.1 Speaker recognition component interface

The Speaker Recognition component exposes the *ISpeakerRecognition* interface to other services and applications. The interface provides seven capabilities for handling speakers (create, query, delete, enroll) and recognize or identify enrolled ones.

1.3.1.1 Capability *ISpeakerRecognition::createUser()*

Syntax

string createUser(string userID)

Description

This method is used for creating anew user and registering in speakers DB.

Example

string createUser("user01")

In this example the createUser method is called for creating and registering user *user01*. Upon a successful creation an example response string will be:

```
<?xml version="1.0" ?>
<speakerRecognition>
<returnCode> 0 </returnCode>
<returnMessage> user 'user01' created successfully </returnMessage>
</speakerRecognition>
```

1.3.1.2 Capability *ISpeakerRecognition::queryUser()*

Syntax

string queryUser(string userID)

Description

This method is used for gathering information about a user.

Example

string queryUser("user01")

In this example the queryUser method is called for getting information about user *user01*. An example response string will be:

```
<?xml version="1.0" ?>
<speakerRecognition>
<returnCode> 0 </returnCode>
<returnMessage> user 'user01' exists </returnMessage>
</speakerRecognition>
```

1.3.1.3 Capability *ISpeakerRecognition::clearUser()*

Syntax

string clearUser(string userID)

Description

This method is used for deleting the voiceprint of a user without de-registering him from speakers DB.

Example

```
string clearUser("user01")
```

In this example the clearUser method is called for clearing user *user01* voiceprint. Upon a successful execution an example response string will be:

```
<?xml version="1.0" ?>
<speakerRecognition>
<returnCode> 0 </returnCode>
<returnMessage> user 'user0' cleared successfully</returnMessage>
</speakerRecognition>
```

1.3.1.4 Capability ISpeakerRecognition::deleteUser()*Syntax*

```
string deleteUser(string userID)
```

Description

This method is used for clearing a user's voiceprint and de- registering him from speakers DB.

Example

```
string deleteUser("user01")
```

In this example the deleteUser method is called for deleting user *user01*. Upon a successful creation an example response string will be:

```
<?xml version="1.0" ?>
<speakerRecognition>
<returnCode> 0 </returnCode>
<returnMessage> user 'user0' deleted successfully </returnMessage>
</speakerRecognition>
```

1.3.1.5 Capability ISpeakerRecognition::enrollUser()*Syntax*

```
string enrollUser(string userID, string utteranceURL, string utteranceText, string
recognitionType)
```

Description

This method is used for enrolling a new user and creating his voiceprint.

Example

```
string enrollUser("user01", "C:\SpeakerRecognition\enrollment\utte_user01.pcm", "", "normal" )
```

In this example the enrollUser method is called for creating the voiceprint of user *user01*, using the recorded utterance *utte_user01.pcm*, with *normal* recognition type (text-independent) . Upon a successful creation an example response string will be:

```
<?xml version="1.0" ?>
<speakerRecognition>
<returnCode> 0 </returnCode>
<returnMessage> user 'user0' enrolled successfully /returnMessage>
</speakerRecognition>
```

1.3.1.6 Capability ISpeakerRecognition::verifyUser()

Syntax

```
string verifyUser(string userID, string utteranceURL, string utteranceText, string  
recognitionType, bool modelAdaptation)
```

Description

This method is used to verify a user's identity.

Example

```
string verifyUser("user01", "C:\SpeakerRecognition\enrollment\ver_utte_user01.pcm", "",  
"normal", false )
```

In this example the verifyUser method is called for verifying user *user01* using the recorded utterance *ver_utte_user01.pcm*, with *normal* recognition type (text-independent) and without performing voiceprint model adaptation . Upon a successful creation an example response string will be:

```
<?xml version="1.0" ?>  
<speakerVerification>  
<returnCode> 0 </returnCode>  
<verificationScore> 785 </verificationScore>  
</speakerVerification>
```

1.3.1.7 Capability ISpeakerRecognition::identifyUser()

Syntax

```
string identifyUser(string utteranceURL, string utteranceText, string recognitionType)
```

Description

This method is used for identifying a user.

Example

```
string identifyUser("C:\SpeakerRecognition\enrollment\ident_utte_user01.pcm", "", "normal")
```

In this example the identifyUser method is called for identifying who is the user *using* the recorded utterance *ident_utte_user01.pcm*, with *normal* recognition type (text-independent). Upon a successful creation an example response string will be:

```
<?xml version="1.0" ?>  
<speakerIdentification>  
<returnCode> 0 </returnCode>  
<userID> user01 </userID>  
<identificationScore> 950 </identificationScore>  
</speakerIdentification>
```


1.3.2 Explicit speech interaction component interface

This component exposes to other services and application services the *IVoiceIO* interface to provide three main functionalities: recognize and understand user input, synthesize predefined system outputs and generate system responses. The capabilities offered through this interface are described in the following sections, including descriptions and examples of use.

1.3.2.1 Capability *IVoiceIO::ListenToUser()*

Syntax

```
string listenToUser(string userID, string applicationID, string sessionID, string grammarURL,
                  string semVar, string speechInputURL)
```

Description

This method is used for accessing Speech recognition and understanding component. It requires *userID*, *applicationID* and *sessionID* as input for setting up the proper resources, the URL of the grammar to be used for the current speech understanding and the semantic variable where the semantics are stored, and finally the URL of the recorded speech input. It returns a string containing information for the recognized and understood data formatted in a XML schema.

Example

```
string result = proxy.listenToUser("myUser", "myApp", "mySession",
                                   "C:\\myApp\\GetAction.grxml", "command",
                                   newrecFileURL);
```

In this example the *ListenToUser* method is called from *myApp* application for user *myUser* and session *mySession*. The system will try to recognize and understand the user sentence contained in *newrecFileURL* using the grammatical and semantic information off *GetAction.grxml* grammar. A sample grammar follows (for more detailed information on GRXML grammar syntax refer to D4.2 §5.3.2: Speech Recognition Grammar Specification).

```
<?xml version="1.0"?>
<grammar xml:lang="en-us" version="1.0" xmlns="http://www.w3.org/2001/06/grammar" root="rootrule">

<meta name="maxspeechtimeout" content="5000"/>
<meta name="incompletetimeout" content="2000"/>

  <rule id="rootrule" scope="public">
    <tag>command="";</tag>
    <item repeat="0-1"><ruleref uri = "#prefix"/></item>
    <one-of>
      <item><ruleref uri= "#lights_on"/> <tag>command = 'LIGHTS_ON'</tag></item>
      <item><ruleref uri= "#lights_off"/> <tag>command = 'LIGHTS_OFF'</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri = "#suffix"/></item>
  </rule>

  <rule id="prefix">
    <one-of>
      <item>I want to</item>
      <item>I would like to</item>
      <item>Please</item>
    </one-of>
```

```

</rule>

<rule id="suffix">
  <one-of>
    <item>Please</item>
  </one-of>
</rule>

<rule id="lights_on">
  <one-of>
    <item>turn on the lights</item>
    <item>turn the lights on</item>
    <item>switch on the lights</item>
  </one-of>
</rule>

<rule id="lights_off">
  <one-of>
    <item>turn off the lights</item>
    <item>turn the lights off</item>
    <item>switch off the lights</item>
  </one-of>
</rule>

</grammar>

```

The recognition and understanding result is returned in *result* string to the calling application or service. For the above mentioned example *result* would contain the following:

```

<?xml version="1.0" ?>
- <speechRecognition>
  <returnCode>0</returnCode>
  - <returnMessage>
    <semanticValue>LIGHTS_ON</semanticValue>
    <confidence>95</confidence>
    <utterance>Turn the lights on</utterance>
  </returnMessage>
</speechRecognition>

```

Two error situations can happen:

- no speech found in the sound file sent for recognition (the user did not speak or the environment is so noisy that the speech can not be separated from surround noisy). In that case a NOINPUT is returned as recognition result, depicted in the following example.

```

<?xml version="1.0" ?>
- <speechRecognition>
  <returnCode>0</returnCode>
  - <returnMessage>
    <semanticValue>NOINPUT</semanticValue>
    <confidence />
    <utterance />
  </returnMessage>
</speechRecognition>

```

- the user spoke a not understood sentence or word (out of the certain grammar). In that case a NOMATCH is returned as recognition result, depicted in the following example.

```
<?xml version="1.0" ?>
- <speechRecognition>
  <returnCode>0</returnCode>
  - <returnMessage>
    <semanticValue>NOMATCH</semanticValue>
    <confidence />
    <utterance />
  </returnMessage>
</speechRecognition>
```

1.3.2.2 Capability IVoiceIO::ListenToUserBL()

Syntax

```
string listenToUserBL(string userID, string applicationID, string sessionID, string
grammarPath, string speechInputURL);
```

Description

This method is similar with the *listenToUser()* but it returns all recognition results in plain format. The recognizer is able to return the n-best matching results for certain recognition. The developer has to parse the return message and select the best fit to the application result.

Example

```
string result = proxy.listenToUser("myUser", "myApp", "mySession",
                                   "C:\\myApp\\GetAction.grxml", newrecFileURL);
```

An example *result* would contain the following (2 recognition candidates):

```
<?xml version='1.0' ?>
<result>
  <interpretation grammar="Digits_Grammar" confidence="98">
    <input mode="speech">switch on the lights please</input>
    <instance>
      <command confidence="98">LIGHTS_ON</command>
      <SWI_literal>switch on the lights please</SWI_literal>
      <SWI_grammarName>GetAction.grxml</SWI_grammarName>
    </instance>
  </interpretation>
  <interpretation grammar="Digits_Grammar" confidence="14">
    <input mode="speech">switch off the lights</input>
    <instance>
      <command confidence="14">LIGHTS_OFF</command>
      <SWI_literal>switch off the lights</SWI_literal>
      <SWI_grammarName>GetAction.grxml</SWI_grammarName>
    </instance>
  </interpretation>
</result>
```

1.3.2.3 Capability IVoiceIO::interact()*Syntax*

```
string interact(string userID, string applicationID, string sessionID,
               string grammarURL, string semVar,
               string promptURL, string language);
```

Description

This method combines both speech output and speech input functionality for a single dialogue step. It automatically uses `speakToUser` to prompt the user for an action and then captures and recognizes user's input. The input parameters are similar to the previous methods. There are two more input parameters: *promptURL* is the path for the prompt to be synthesized and spoken and *language* is the language to be used in `SpeakToUser` (see description of that capability later on).

Example

```
string result = proxy.interact("myUser", "myApp", "mySession", "C:\\myApp\\GetAction.grxml",
                              "command",
                              "C:\\myApp\\what_to_do.txt", "en-US");
```

In this example the service will firstly synthesize and play the prompt stored in file *what_to_do.txt* and then it will capture user input and recognize it against *GetAction.grxml* grammar. The format of the *result* is as described in `IVoiceIO::ListenToUser` method.

1.3.2.4 Capability IVoiceIO::interactBL()*Syntax*

```
string interactBL(string userID, string applicationID, string sessionID,
                  string grammarURL,
                  string promptURL, string language);
```

Description

Similar to the `IVoiceIO::ListenToUserBL` this method is used for returning the n-best recognition results. The semantic variable is not passed as input parameter.

Example

```
string result = proxy.interact("myUser", "myApp", "mySession", "C:\\myApp\\GetAction.grxml",
                              "C:\\myApp\\what_to_do.txt", "en-US");
```

The format of the *result* is as described in `IVoiceIO::ListenToUserBL` method.

1.3.2.5 Capability IVoiceIO::interactEH()*Syntax*

```
string interactEH(string userID, string applicationID, string sessionID,
                  string ApplResourcesPath, string grammarURL, string semVar,
                  string promptURL, string notUnderstoodPromptURL,
                  string noInputPromptURL,
                  int totalRepeats, int confidenceLimit);
```

Description

This method provides the most advance and automated interaction; it provides a complete speech-based dialogue interaction act in one call from the application. It uses the *userID* to contact UMPS service and find the user's preferences concerning language and speech

parameters. Then the entire speech interface is adapted to those preferences (in case of no such user exists or the UMPS is not accessible the default settings are used). Then the `IVoiceIO::interact()` is called to perform a simple interaction step. In case of success recognition and understand the method finish returning a *result* similar to the `IVoiceIO::ListenToUser` method. In case of error situations an error-handling sub-dialogue is initiated: the *notUnderstoodPromptURL* or *noInputPromptURL* prompt is synthesized and played and a new dialogue cycle starts until there is a success recognition or the maximum number of *totalRepeats* reached. The same process takes place in case there is a valid speech input and recognition but the recognition confidence is below a threshold defined by the *confidenceLimit* input parameter.

Example

```
String result = interactEH("myUser", "myApp", "mySession",
    "C:\\MyApp\\ApplicationResources", "GetAction.grxml", "command",
    "What_to_do.txt", "no_input.txt", "no_understood.txt", 3, 60);
```

In this example the service will prompt user with *What_to_do.txt* prompt and try to recognize and understand his input against *GetAction.grxml* grammar. If NOINPUT situation occurs the system will synthesize and play the *no_input.txt* prompt and restart the dialogue from the begin. Similarly in case of NOMATCH or low confidence the system will synthesize and play the *no_understood.txt* prompt. The dialogue will cycle for 3 times maximum and the confidence threshold is set to 60 (in a scale of 0-100).

Important Note: In order the service to be able to adapt to different languages the grammar and the prompts are not passed with the entire path but with the simple filenames. Refer to the VoiceIO tutorial for a multilingual application structure.

The format of the *result* is as described in `IVoiceIO::ListenToUserBL` method.

1.3.2.6 Capability `IVoiceIO::SpeakToUser()`

Syntax

```
string speakToUser(string userID, string applicationID, string promptText, string promptURL,
    string language);
```

Description

This method is used for accessing Response Synthesis component. It requires `userID` and `applicationID` as input for setting up the proper resources as well as the string with the text to be synthesized or the path of the file containing the text of the response to be synthesized and played to user and the language of the prompt. It returns a string containing the path of the synthesized response.

Example

```
String myResponse SpeakToUser("user01", "EntranceManager", "",
    "C:\\EntranceManager\\responses\\intro_morning.txt", "en-US")
```

In this example the `SpeakToUser` method is called from `EntranceManager` application for user `user01`. The system will synthesize the text contained in *intro_morning.txt* file.

```
String myResponse SpeakToUser("user01", "EntranceManager", "Good morning. How can I
    help you?", "", "en-US")
```

In this example the `SpeakToUser` method is called from `EntranceManager` application for user `user01`. The system will synthesize the text contained in the string *“Good morning. How can I help you?”*.

The path of the synthesized response file is returned in `myResponse` string to the calling application or service.

The following table summarizes the supported languages available for Amigo project and the related male and female users in UMPS:

| Symbol | Language | Spoken in | Female | Mail |
|--------|----------|---------------|----------|------------|
| de-DE | German | Germany | Steffi | Thomas |
| en-US | English | United States | Maria | Jerry |
| es-ES | Spanish | Spain | Isabel | Pedro |
| fr-FR | French | France | Virginie | Nikola |
| it-IT | Italian | Italy | Silvia | Paolo |
| nl-NL | Dutch | Netherlands | Claire | Herman |
| el-GR | Greek | Greece | Afroditi | Alexandros |
| eu-ES | Basque | Spain | Arantxa | Imanol |

1.3.2.7 Capability `IVoiceIO::SpeakToUserAdv()`

Syntax

```
string speakToUserAdv(string userID, string applicationID, string ApplResourcesPath, string promptURL);
```

Description

The advance `SpeakToUser` method uses the `userID` in order to contact UMPS and get the user's preferred language. Then it uses the proper language recourses for the specific language to produce the prompt in the desired language. That's why the `ApplResourcesPath` is required now as input parameter.

Example

```
String myResponse SpeakToUserAdv("user01", "EntranceManager", "",
                                "C:\\EntranceManager\\ApplicationResources\\",
                                "intro_morning.txt")
```

1.3.2.8 Capability `IVoiceIO::GenerateResponse()`

Syntax

```
string GenerateResponse(string nlgTemplateURL)
```

Description

This method is used for accessing Response Generation module. The URL of the `nlgTemplate` to be used for the generation should be passed as input in each call. The method returns the path of the file that contains the generated response in SSML format.

Example

```
String myResponse GenerateResponse("intro_morning.grxml")
```

In this example the `GenerateResponse` method is called from `EntranceManager` application for generating response based on *intro_morning.grxml* template. The SSML formatted response is stored in *intro_morning.ssml* and can be used in a later state from speech synthesis engine to synthesize and play the response.

1.4 Tutorial

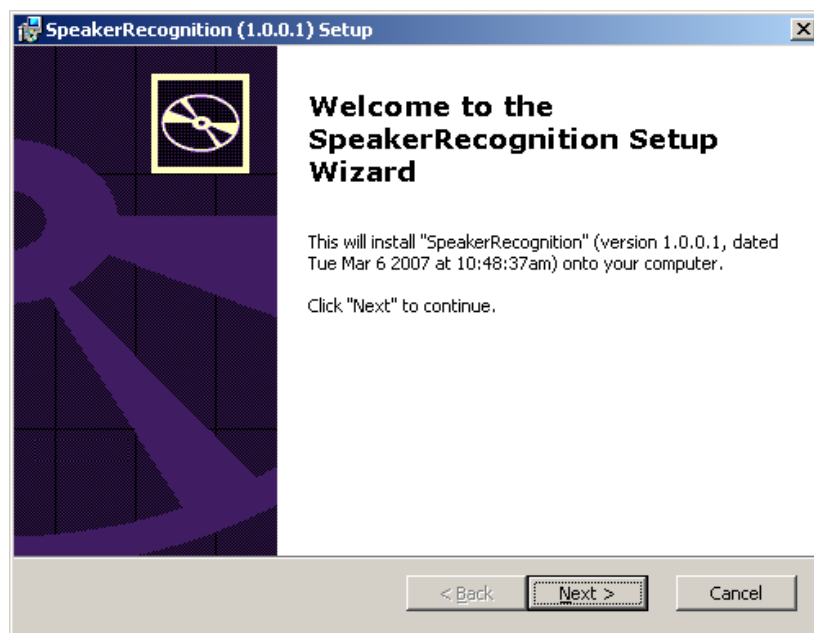
1.4.1 SpeakerRecognition Service

1.4.1.1 Install SpeakerRecognition

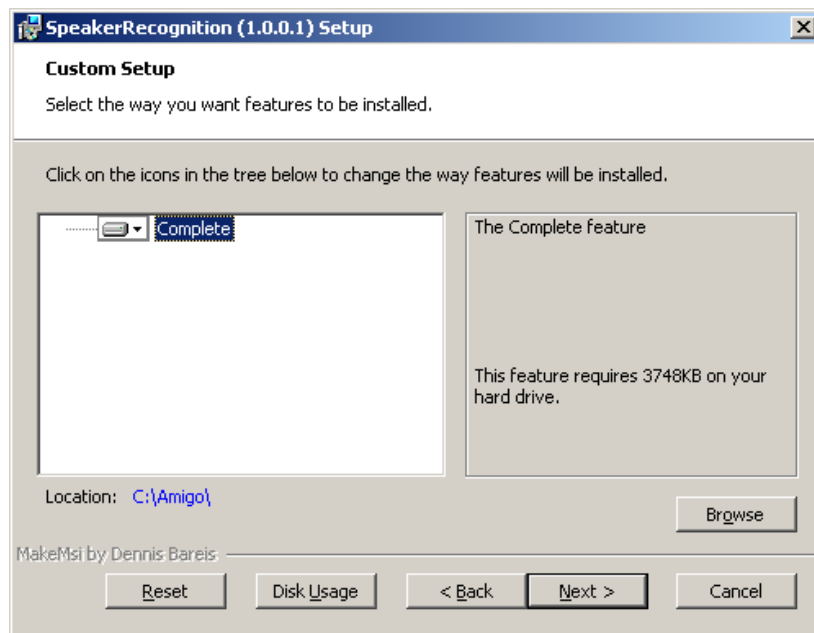
SpeakerRecognition service, demo applications, background models, demo speaker voiceprints and all required configuration files and libraries are provided as an msi installation package for Windows platforms.

Step 1

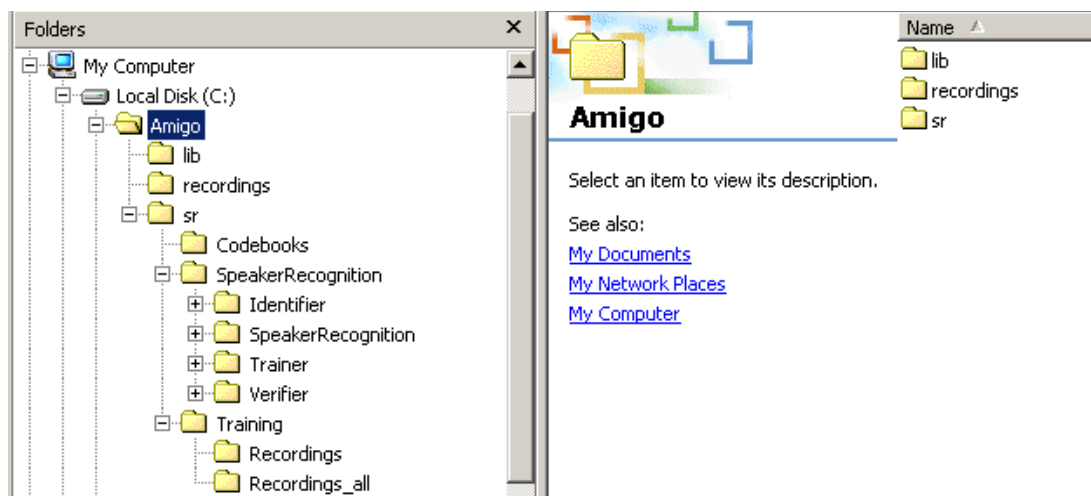
Execute ***Speaker Recognition.msi*** and follow the instructions of the installation package.



It is strongly recommended to accept the default installation path: C:\Amigo\; otherwise you have to manually update the configuration files.



Assuming that you accepted the default installation folder then the following structure will be created:



Folder **C:\Amigo** contains the subfolders:

- lib -> the folder where the required libraries (dlls) are stored
- recordings -> the folder where the recorder speech files are stored
- sr -> the folder where the Speaker Recognition service and demo applications are stored

Folder **C:\Amigo\lib** contains the files:

- EMIC.FirewallHandler.dll -> Amigo .Net framework

- EMIC.WebServerComponent.dll -> Amigo .Net framework
- EMIC.WSAddressing.dll -> Amigo .Net framework
- kptrnng.dll -> library for speaker recognition engine licensing
- MyWinAPI -> library for ini and configuration file manipulation
- SpkRec.dll -> speaker recognition engine
- WaveIO.dll -> library for microphone / RTP capture and playing sound files (produced by WaveIO solution)
- WSDiscovery.Net.dll -> Amigo .Net framework

Folder **C:\Amigo\recordings** is empty at the begin

Folder **C:\Amigo\lsr** contains:

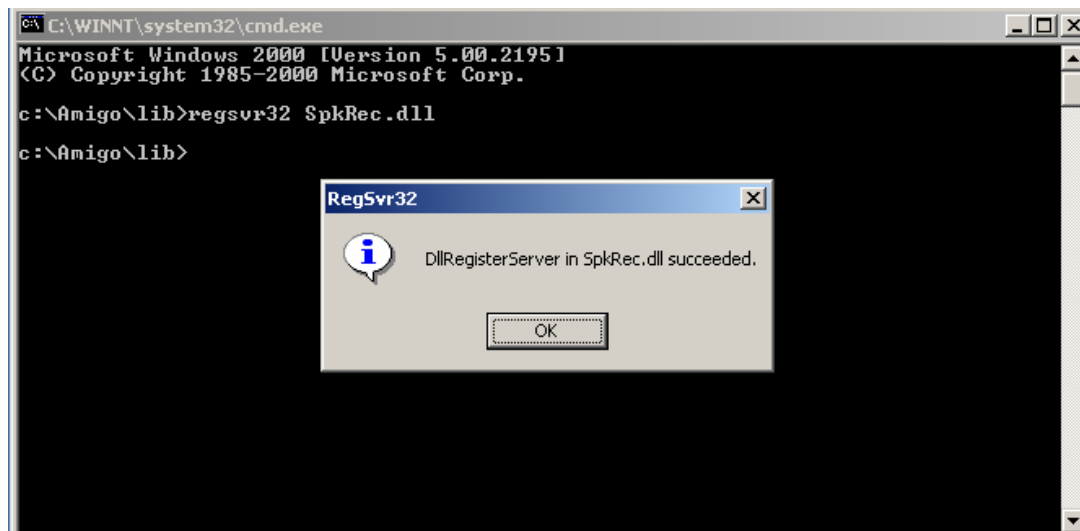
- Codebooks -> the folder where voiceprints are stored
- SpeakerRecognition -> the folder of SpeakerRecognition, Trainer, Verifier and Identifier C# projects
- Training -> the folder where the collected data for training are stored
- SpeakerWebServices.cfg, SpkSettings.cfg, TrainingPromts.txt-> configuration and initialization files

Step 2

Register the SpkRec.dll library. From command line enter the following command:

>regsvr32 SpkRec.dll.

Attention: The license library kptrnng.dll should be present in the same folder with SpkRec.dll or in the System32 folder of Windows. A time-limited license is provided until the end of the project.



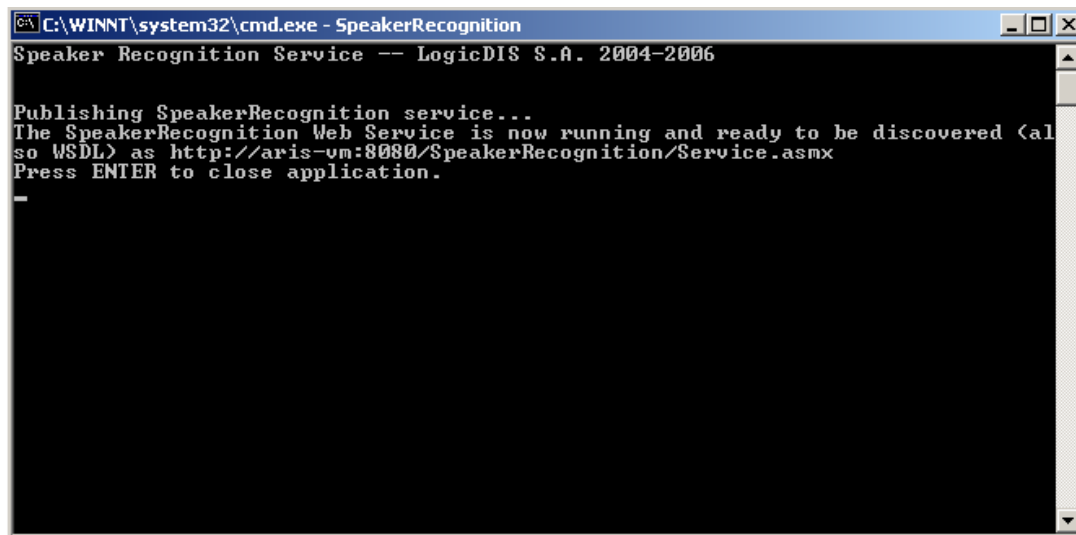
Step 3

Rebuild SpeakerRecognition solution if you want to change the settings for server name and port (default 8080).

1.4.1.2 Run SpeakerRecognition Server

Launch the SpeakerRecognition service by executing the **SpeakerRecognition.exe** located in: **C:\Amigo\src\SpeakerRecognition\SpeakerRecognition\bin\Release**

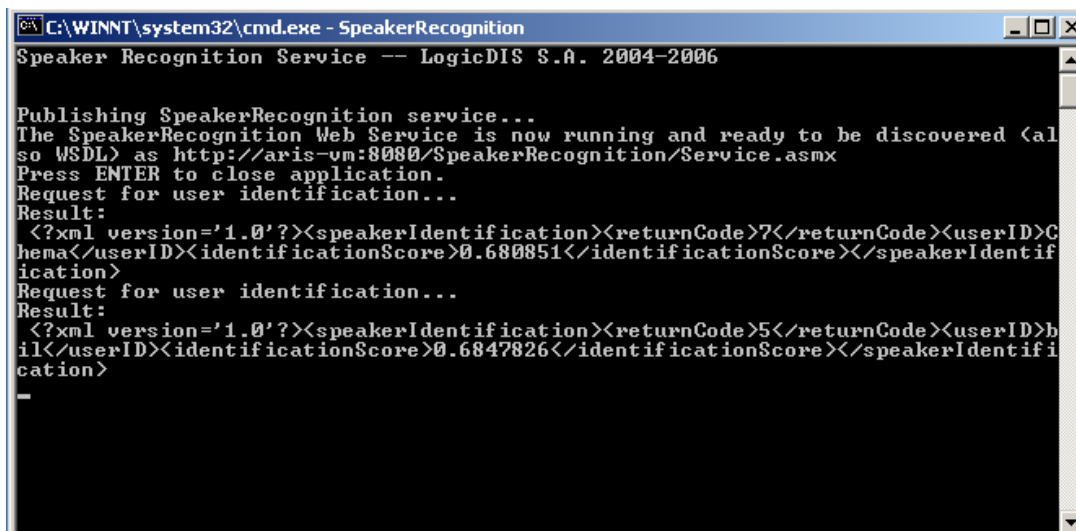
On successful execution the service will be published and ready to accept requests for identification / verification tasks.



```
C:\WINNT\system32\cmd.exe - SpeakerRecognition
Speaker Recognition Service -- LogicDIS S.A. 2004-2006

Publishing SpeakerRecognition service...
The SpeakerRecognition Web Service is now running and ready to be discovered (also WSDL) as http://aris-vm:8080/SpeakerRecognition/Service.asmx
Press ENTER to close application.
-
```

Every time a request for speaker recognition arrives the console displays information regarding the identification / verification process.



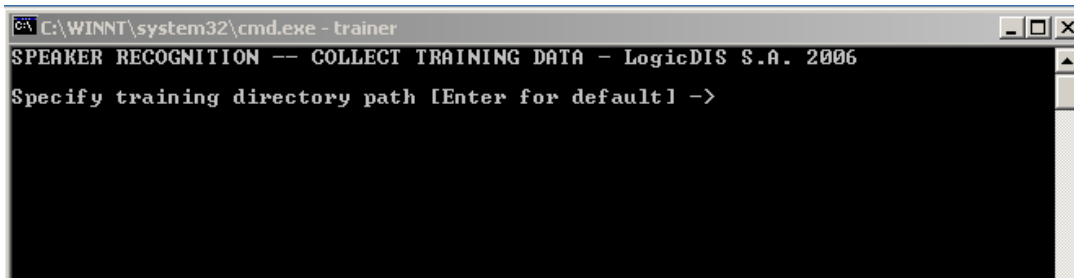
```
C:\WINNT\system32\cmd.exe - SpeakerRecognition
Speaker Recognition Service -- LogicDIS S.A. 2004-2006

Publishing SpeakerRecognition service...
The SpeakerRecognition Web Service is now running and ready to be discovered (also WSDL) as http://aris-vm:8080/SpeakerRecognition/Service.asmx
Press ENTER to close application.
Request for user identification...
Result:
<?xml version='1.0'?><speakerIdentification><returnCode>7</returnCode><userID>C hema</userID><identificationScore>0.680851</identificationScore></speakerIdentification>
Request for user identification...
Result:
<?xml version='1.0'?><speakerIdentification><returnCode>5</returnCode><userID>b il</userID><identificationScore>0.6847826</identificationScore></speakerIdentification>
-
```

1.4.1.3 Run Trainer Demo Application

Trainer application is implemented to demonstrate the use of SpeakerRecognition service for enrolling a new speaker's voiceprint: in order the SpeakerRecognition service to be able to identify or verify a speaker's identity a model of that speaker should be created and stored in the **C:\Amigo\Codebooks** folder. The model (voiceprint) is produced by processing an number of speech samples of the particular user. In this demo we use 10 sentences/files for creating the voiceprint plus 4 additional sentences/files for calculating the thresholds for the particular user.

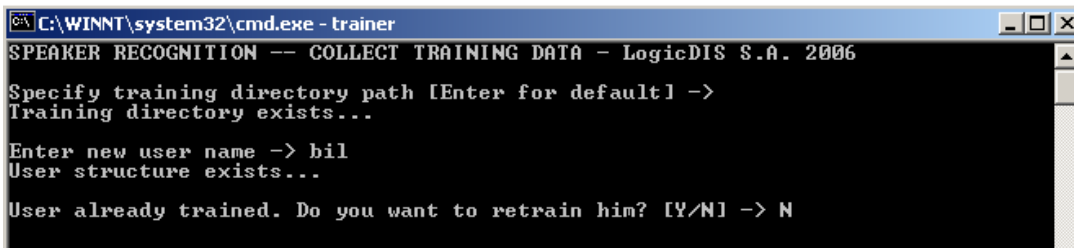
Launch the application by executing the **Trainer.exe** located in **C:\Amigo\sr\SpeakerRecognition\Trainer\bin\Release**



```
C:\WINNT\system32\cmd.exe - trainer
SPEAKER RECOGNITION -- COLLECT TRAINING DATA - LogicDIS S.A. 2006
Specify training directory path [Enter for default] ->
```

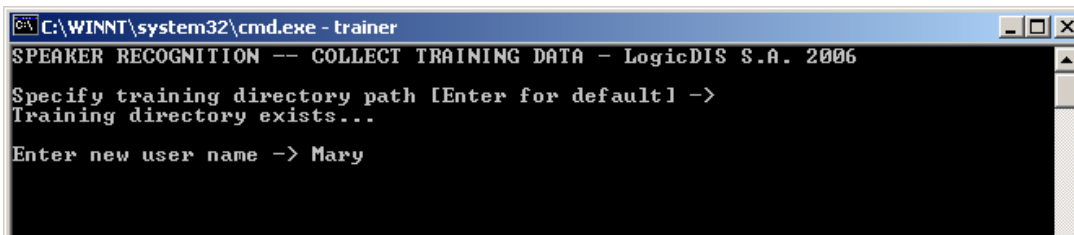
The application is asking user to specify the path for the directory that the training speech input will be stored. The default directory is: *C:\Amigo\sr\Training* and is defined in the configuration file *SpkSettings.cfg*

Then the application asks for new speaker name (userID). If the specified userID exists in the enrolled speakers' database the application confirms that the user wish to re-train the particular speaker.

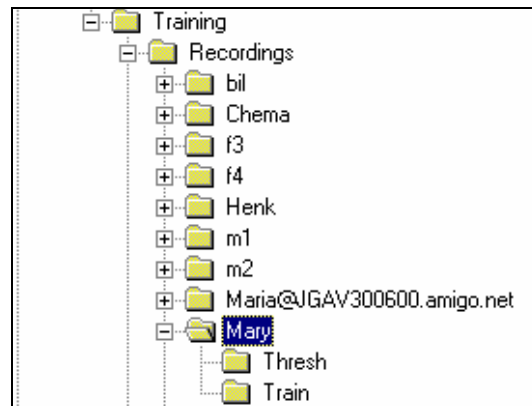


```
C:\WINNT\system32\cmd.exe - trainer
SPEAKER RECOGNITION -- COLLECT TRAINING DATA - LogicDIS S.A. 2006
Specify training directory path [Enter for default] ->
Training directory exists...
Enter new user name -> hil
User structure exists...
User already trained. Do you want to retrain him? [Y/N] -> N
```

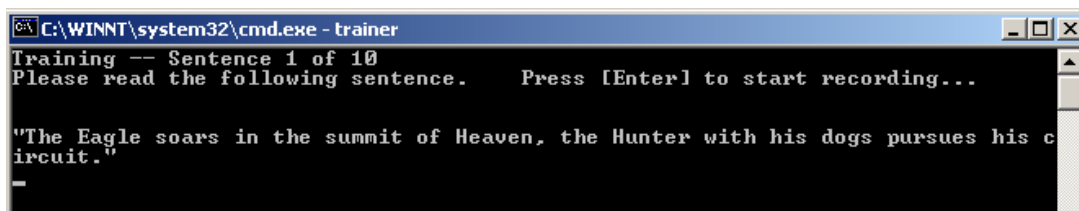
In the opposite situation a structure is created for the new speaker including the folders *userID\Thresh* and *userID\Train* under the *Training\Recordings* directory.



```
C:\WINNT\system32\cmd.exe - trainer
SPEAKER RECOGNITION -- COLLECT TRAINING DATA - LogicDIS S.A. 2006
Specify training directory path [Enter for default] ->
Training directory exists...
Enter new user name -> Mary
```

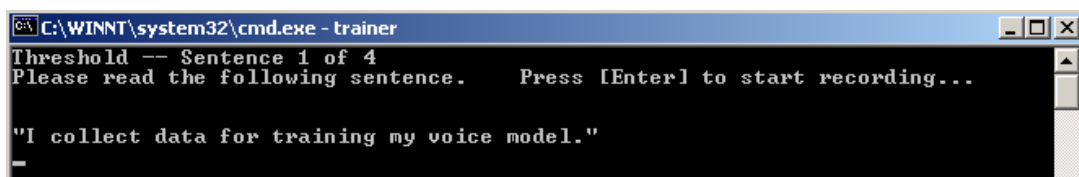


Then the collection of training data is starting. The Application displays a predefined phrase and asks user to repeat it while it records his speech and creates a audio file in the *userId\Train* folder named after the userID and the particular phrase number: i.e. mary1.pcm to Mary10.pcm.

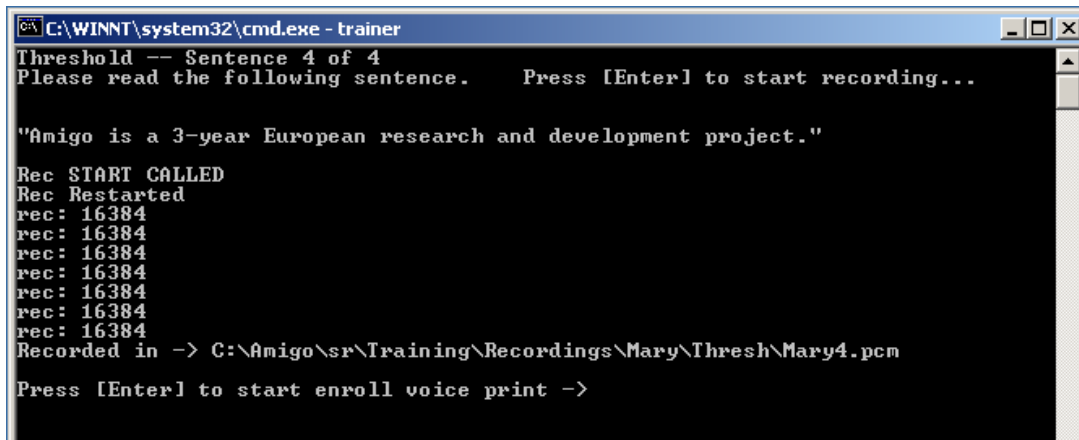


The training sentences as well as the threshold sentences are defined in the *C:\Amigo\src\TrainingPrompts.txt* file. For the purposes of this demo the “Cycles of Heaven” of Edgar Allan Poe is used.

When all 10 training sentences are collected the user is asked to speak additional 4 sentences for thresholds calculation. These sentences are stored *userId\Thresh* folder named after the userID and the particular phrase number: i.e. mary1.pcm to Mary4.pcm.



When the process of collecting threshold data also finishes the application asks confirmation for starting produce the new voice print model and the thresholds for the particular user.



```

C:\WINNT\system32\cmd.exe - trainer
Threshold -- Sentence 4 of 4
Please read the following sentence.  Press [Enter] to start recording...

"Amigo is a 3-year European research and development project."

Rec START CALLED
Rec Restarted
rec: 16384
rec: 16384
rec: 16384
rec: 16384
rec: 16384
rec: 16384
rec: 16384
Recorded in -> C:\Amigo\sr\Training\Recordings\Mary\Thresh\Mary4.pcm
Press [Enter] to start enroll voice print ->

```

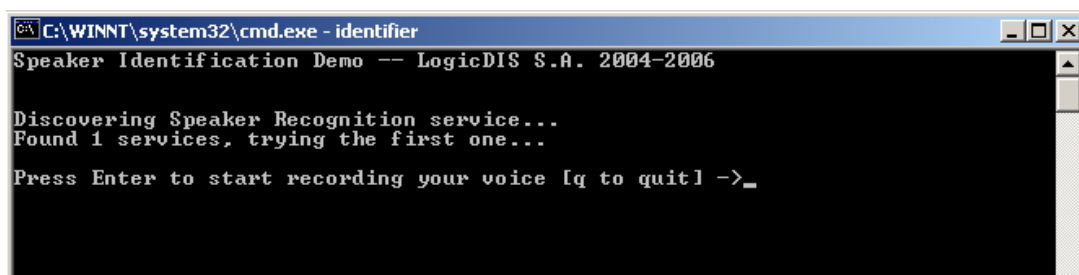
On a successful voice print production the model of the new speaker is added in the *C:\Amigo\sr\Codebooks* folder and information for this speaker is included in the *SpkSettings.cfg* configuration file. For now on the new speaker voiceprint can be used for performing verification and identification tasks as described in the next sections.

1.4.1.4 Run Identifier Demo Application

Identifier application is implemented to demonstrate the use of SpeakerRecognition service for identifying a speaker's identity.

Launch the application by executing the **Identifier.exe** located in **C:\Amigo\sr\SpeakerRecognition\Identifier\bin\Release**.

SpeakerRecognition service should be run because the application tries to discover the service and use it.



```

C:\WINNT\system32\cmd.exe - identifier
Speaker Identification Demo -- LogicDIS S.A. 2004-2006

Discovering Speaker Recognition service...
Found 1 services, trying the first one...

Press Enter to start recording your voice [q to quit] ->_

```

As soon as the application finds the SpeakerRecognition service it starts a loop, asking user to record some speech input (i.e. one utterance). The *WaveIO.dll* library is used for capturing a 6 seconds speech input and storing it into the *C:\Amigo\recordings* directory with a unique file name:

```
recFileURL = myWaveIO.CaptureMic(6);
```

```

Press Enter to start recording your voice [q to quit] ->
Say something....

Rec START CALLED
Rec Restarted
rec: 16384
rec: 16384
rec: 16384
rec: 16384
rec: 16384
rec: 16384
Recorded in -> C:\Amigo\recordings\2007_3_5__13_50_46.raw

```

Then, the SpeakerRecognition service is called for identification on the recorded speech input.

```
xmlIdentificationResult = proxy.identifyUser(recFileURL, "", "normal");
```

The result is returned in a XML formatted string that should be parsed for extracting meaningful information. Two methods are provided for that reason:

```

static public double xmlGetNumNode(string xmlIdentResult, string nodeName)
static public string xmlGetStringNode(string xmlIdentResult, string nodeName)

```

The structure of the SpeakerRecognition service response is as follows:

```

<?xml version='1.0'?>
<speakerIdentification>
    <returnCode>somereturnCode</returnCode>
    <userID>someuserID</userID>
    <identificationScore>someidentificationScore</identificationScore>
    <Error>some Error description</Error>
</speakerIdentification>

```

The following table summarizes the possible returned codes:

| Speaker Recognition – Identification return codes | | |
|---|---------------|---|
| <i>returnCode</i> | <i>Status</i> | <i>Description</i> |
| > 0 | success | returnCode indicates the number id of the identified speaker. The name of identified speaker is returned by referring to field <i>userX</i> of section <i>[spkUsers]</i> of the configuration file <i>SpkSettings.cfg</i> |
| -1 | error | Input audio file does not exist/corrupted |
| -2 | error | meanStd model file does not exist |
| -3 | error | Background reference codebook does not exist |
| -4 | error | At least one of the user models does not exist |
| -5 | error | Less than 0.5 sec of speech in input audio file |
| -6 | error | No user was identified |
| -7 | error | Invalid configuration file. |

The following images present the response XML string and the parsed output of the demo application on both successful and unsuccessful identification.


```
<?xml version="1.0" ?>
- <speakerIdentification>
  <returnCode>9</returnCode>
  <userID>bil</userID>
  <identificationScore>0.6847826</identificationScore>
</speakerIdentification>
```

```
Calling Speaker Recognition Service for speaker identification...
Identification score [>0 OK, else some error occurred] -> 0.6847826
Identified user -> bil
Press Enter to start recording your voice [q to quit] ->
```

```
<?xml version="1.0" ?>
- <speakerIdentification>
  <returnCode>-5</returnCode>
  <Error>Too short input audio file</Error>
</speakerIdentification>
```

```
Calling Speaker Recognition Service for speaker identification...
Error -> Too short input audio file
Press Enter to start recording your voice [q to quit] ->
```

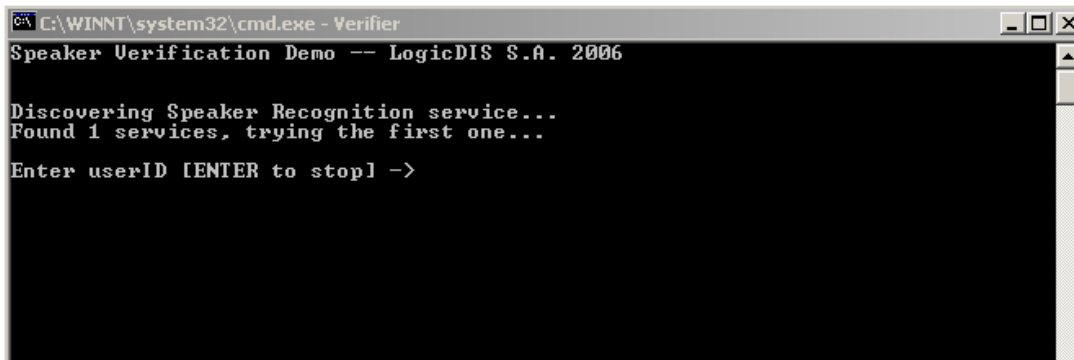
At the end the application asks if the user wants to repeat the entire process or quit.

1.4.1.5 Run Verifier Demo Application

Verifier application is implemented to demonstrate the use of SpeakerRecognition service for verify a speaker's identity.

Launch the application by executing the **Verifier.exe** located in **C:\Amigo\sr\SpeakerRecognition\Verifier\bin\Release**.

SpeakerRecognition service should be run because the application tries to discover the service and use it.

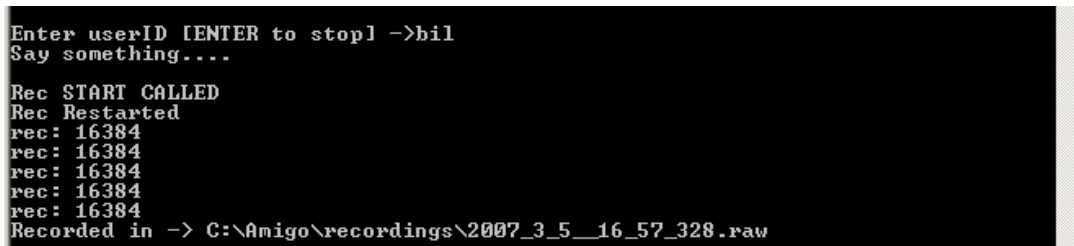


As soon as the application finds the SpeakerRecognition service it starts a loop, asking user to enter the userID and then record some speech input (i.e. one utterance).

The userID will be used to verify that the speaker is really who claims to; it should match one of the userID's stored in the system during the training process.

The *WaveIO.dll* library is used for capturing a 6 seconds speech input and storing it into the *C:\Amigo\recordings* directory with a unique file name:

```
recFileURL = myWaveIO.CaptureMic(6);
```



Then, the SpeakerRecognition service is called for verification on the recorded speech input for the particular userID.

```
xmlVerifResult = proxy.verifyUser(userID, recFileURL, "", "normal", false);
```

The result is returned in a XML formatted string that should be parsed for extracting meaningful information. Two methods are provided for that reason:

```
static public double xmlGetNumNode(string xmlVerResult, string nodeName)
static public string xmlGetStringNode(string xmlVerResult, string nodeName)
```

The structure of the SpeakerRecognition service response is as follows:

```
<?xml version='1.0'?>
<speakerVerification>
  <returnCode>somereturnCode</returnCode>
  <verificationScore>someverificationScore</verificationScore >
  <Error>some Error description</Error>
</speakerVerification>
```

The following table summarizes the possible returned codes:

| Speaker Recognition – Verification return codes | | |
|---|---------------|---|
| <i>returnCode</i> | <i>Status</i> | <i>Description</i> |
| 1 | success | Speaker is verified (accepted) |
| 0 | success | Speaker is not verified (rejected) |
| -1 | Error | Input audio file does not exist/corrupted |
| -2 | Error | User model does not exist |
| -3 | Error | Background reference codebook does not exist |
| -4 | Error | Invalid user models structure |
| -6 | Error | Less than 0.5 sec of speech in input audio file |
| -7 | Error | Invalid configuration file. |

The following images present the response XML string and the parsed output of the demo application on both successful and unsuccessful verification.

```
<?xml version="1.0" ?>
- <speakerVerification>
  <returnCode>1</returnCode>
  <verificationScore>0.494382</verificationScore>
</speakerVerification>
```

```
Calling Speaker Recognition Service...
Verification (built-in threshold, 1=yes, 0=no) -> 1
Verification score -> 0.494382
Enter userID [ENTER to stop] -> _
```



The first screenshot shows an XML document with the following content:

```
<?xml version="1.0" ?>
- <speakerVerification>
  <returnCode>-2</returnCode>
  <Error>User model does not exist</Error>
</speakerVerification>
```

The second screenshot shows a command prompt window with the following text:

```
Calling Speaker Recognition Service...
Error -> User model does not exist
Enter userID [ENTER to stop] -> _
```

At the end the application asks if the user wants to repeat the entire process or quit.

1.4.2 Explicit Speech Interaction Service

1.4.2.1 Install ASR and TTS engines

Explicit Speech Interaction Service (VoicelO server) is built on Nuance (ex ScanSoft) OSR3.0 Automatic Speech Recognition engine and RealSpeak 4.0.1 Synthesis engine. The engines and the appropriate language packs are available for download at <http://www.network.nuance.com/>, <http://www.nuance.com>. Also, there were/will be available on CDs during project technical meetings. The installation packages include detailed documentation instructions for installation.

1.4.2.2 Configure ASR and TTS engines

For the basic functionality we use the standard configuration of speech recognition engine and TTS engine. Configuration guides are included in the installation packages of these engines. Sample configuration files were discussed in chapter 3

1.4.2.3 Install \ Update ASR and TTS license

Assuming you have successfully installed Nuance (ex ScanSoft) Open Speech Recognizer 3.0 ASR and /or RealSpeak 4.0 TTS software using the default installation paths (*c:\Program Files\SpeechWorks\OpenSpeech Recognizer*, *c:\Program Files\ScanSoft\RealSpeak 4.0*). A folder named *c:\flexlm* should have been created in your system.

Step 1

Copy the appropriate license file to that folder:

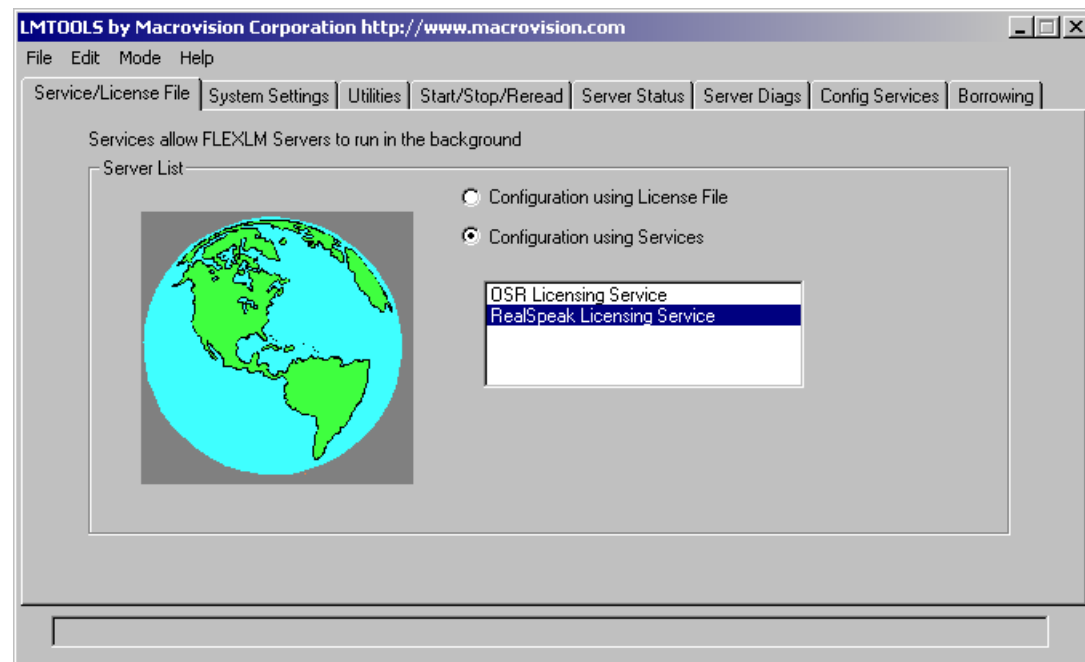
- License file: *asr_license_07-mar-2008.lic* if you intend to use only ASR
- License file: *tts_license_07-mar-2008.lic* if you intend to use only TTS
- License file: *asr+tts_license_07-mar-2008.lic* if you intend to use both ASR and TTS

Step 2

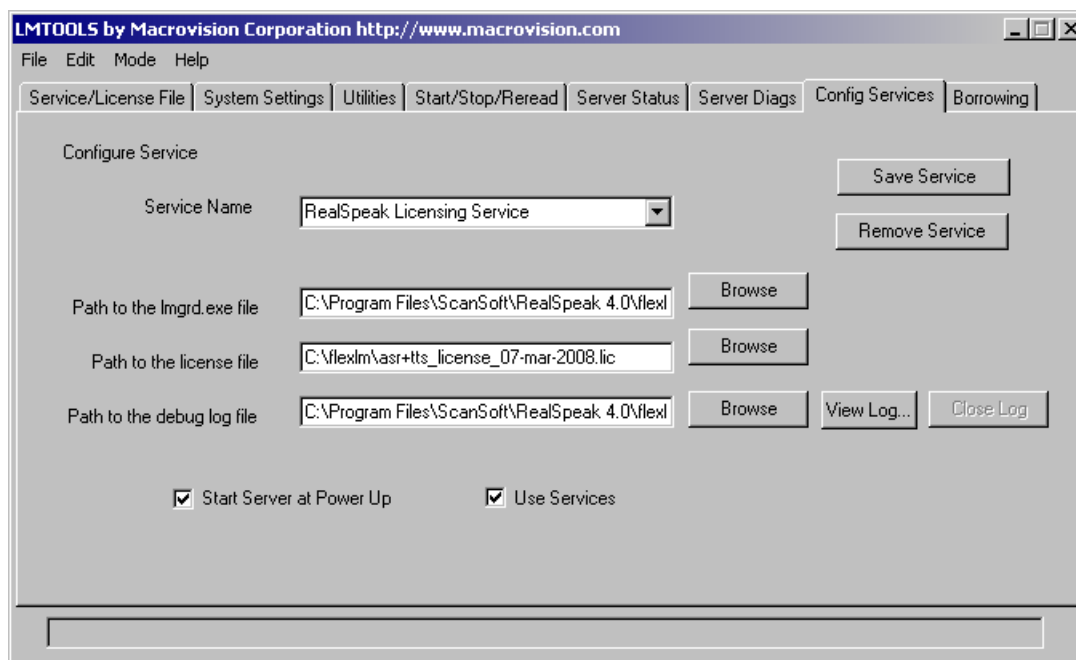
Launch the *Licensing tools* application located at: *Start -> Programs -> Open Speech Recognizer 3.0 -> Licensing Tools* or *Start -> Programs -> Scansoft -> Real Speak 4.0 -> Licensing Tools*

Step3

In the Service/License tab select the RealSpeak Licensing Service

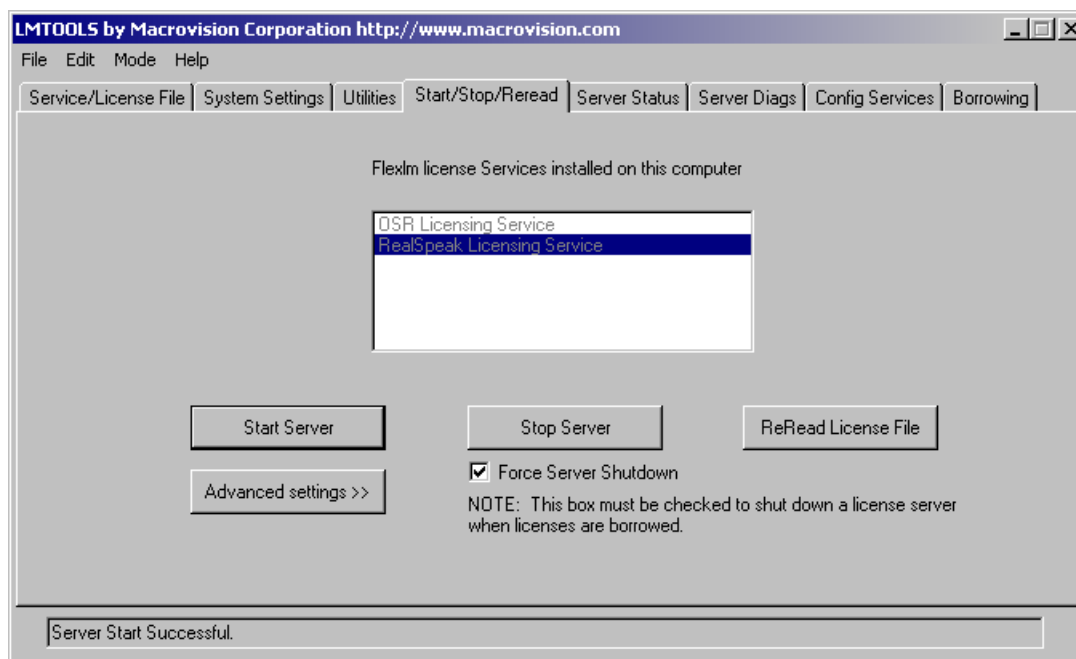
**Step 4**

In the Config Services tab select the valid license file in the "Path to the license file" input field. Make sure that "Start Server at Power Up" and "Use Services" options are both checked.



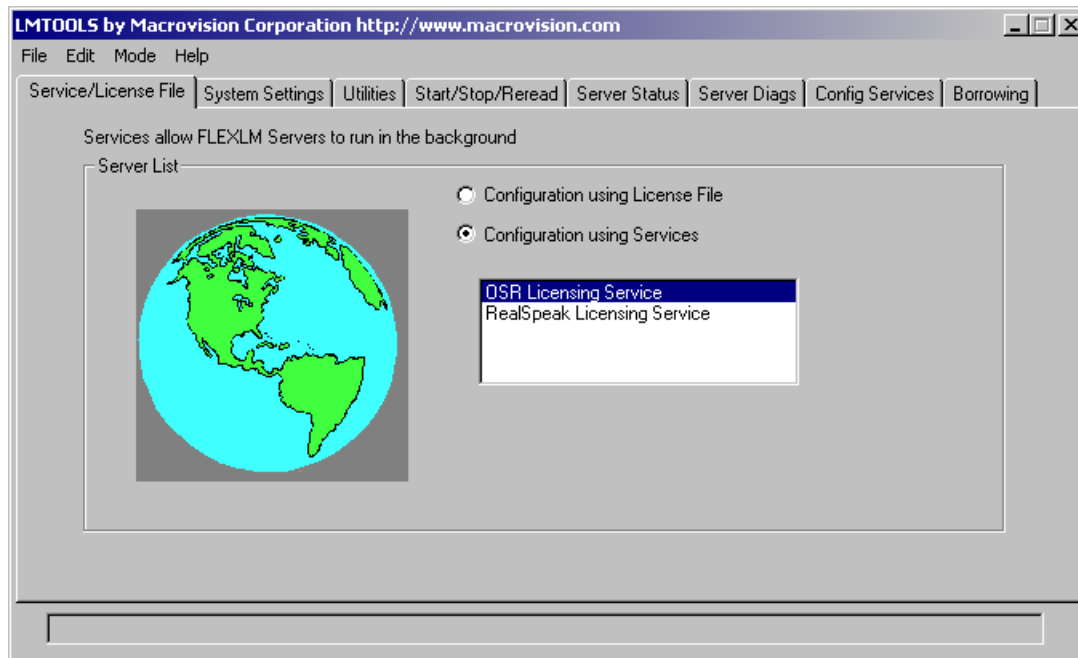
Step 5

In the Start/Stop/Reread tab first stop server with “Force Server Shutdown” option checked. Then Start Server



Step 6

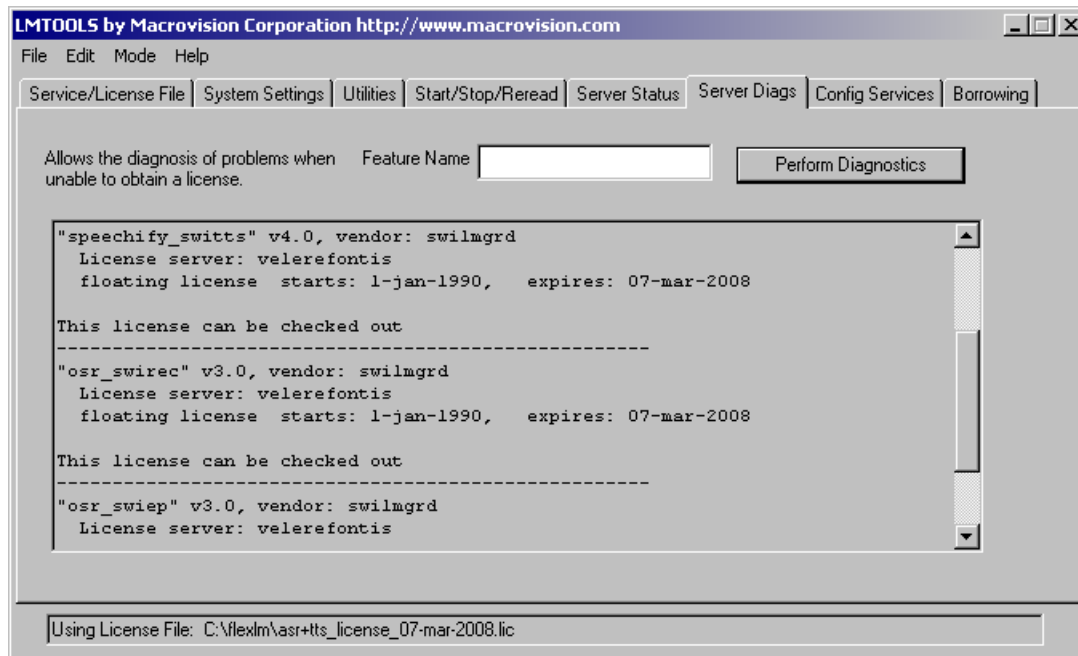
Repeat Steps 3-5 this time selecting the OSR Licensing service



Step 7

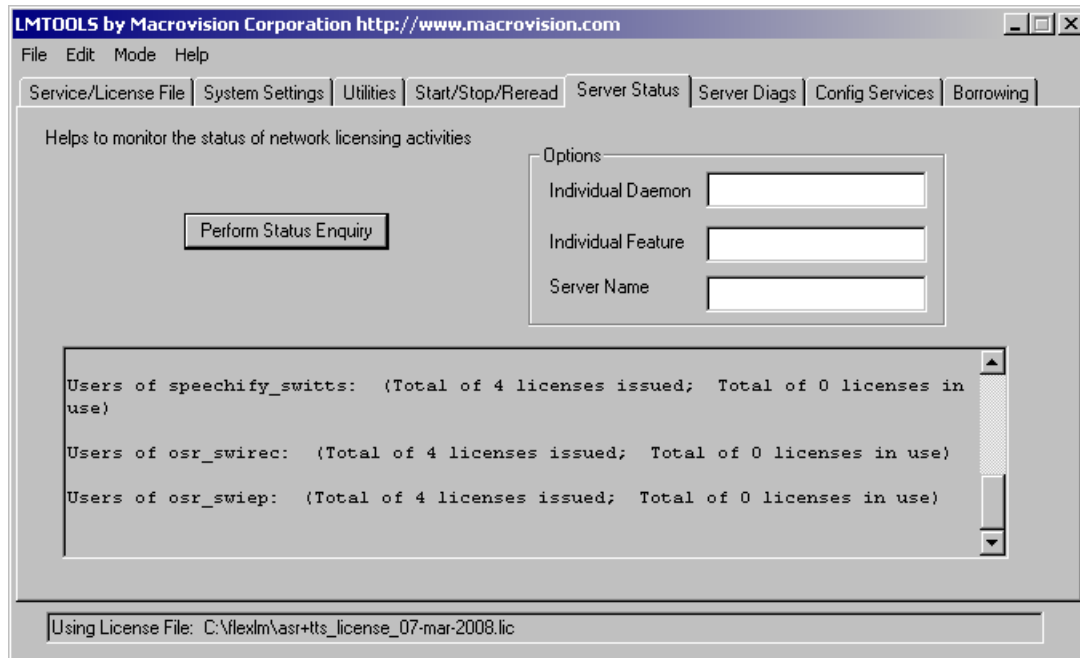
In Server Diags tab press “Perform Diagnostics” and make sure that the appropriate licenses are presented with the comment “This license can be checked out”:

- “speechify_switts” for RealSpeak 4.0 TTS
- “osr_swirec” and “osr_swiep” fro OSR 3.0



Step 8

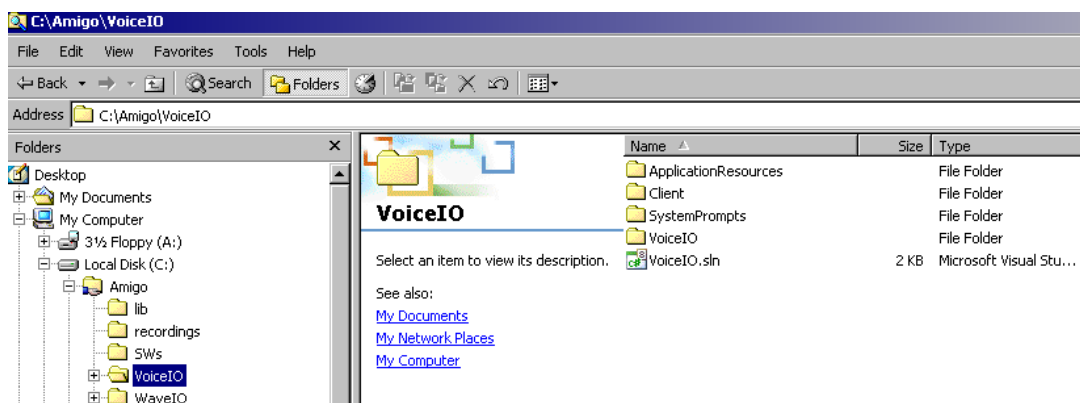
In Server Status tab press “Perform Status Enquiry”. If everything works fine you will have 4 ports for speechify_switts, 4 ports for osr_swirec and 4 ports for osr_swiep



ASR and TTS engines are licensed correctly and ready to be used by VoiceIO service. Please note that VoiceIO service allocates 1 port for TTS and 1 port for ASR thus with the above licenses you are able to use 4 instances of VoiceIO service at the same time in the same machine (assuming that the machine is powerful enough!).

1.4.2.4 Install VoiceIO**Step 1**

Create the folder structure illustrated in the following picture.



Folder **C:\Amigo** contains the subfolders:

- lib -> the folder where the required libraries (dlls) are stored

- recordings -> the folder where the recorder speech files are stored
- SWs -> the folder where the speech files ready for recognition are stored
- VoicelO -> the folder of the VoicelO C# solution
- WavelO -> the folder of the WavelO C# solution

Folder **C:\AmigoVib** contains the files:

- EMIC.FirewallHandler.dll -> Amigo .Net framework
- EMIC.WebServerComponent.dll -> Amigo .Net framework
- EMIC.WSAddressing.dll -> Amigo .Net framework
- OSRRecog.dll -> library for using OSR 3.0 recognition engine
- ttssvr.dll -> library for using RealSpeak 4.0 synthesis engine
- WaveConverter.dll -> library for wave file transformations (produced by WavelO solution)
- WavelO.dll -> library for microphone / RTP capture and playing sound files (produced by WavelO solution)
- WSDiscovery.Net.dll -> Amigo .Net framework

Folder **C:\Amigo\recordings** is empty at the begin

Folder **C:\Amigo\SWs** is empty at the begin

Folder **C:\Amigo\VoicelO** contains:

- VoicelO.sln -> the C# solution file
- VoicelO -> the folder of VocelO server C# project
- Client -> the folder of a demo client application C# project
- ApplicationResources -> the folder contains the grammar files for the demo client application
- SystemPrompts -> the folder the prompt text files for the demo client application

Step 2

Register the BuildSentence.dll library. From command line enter the following command:

>regsvr32 BuildSentence.dll

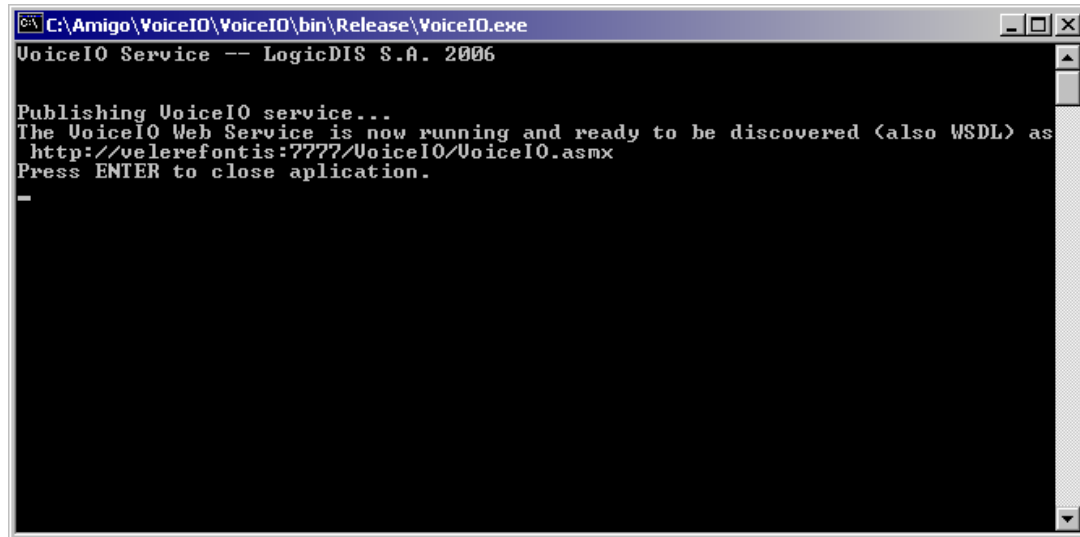
Step 3

Rebuild VoicelO server and client projects if you want to change the settings for server name and port (default 7777).

1.4.2.5 Run VoiceIO Server and Client Application

Step 1

Launch the VoiceIO service by executing the **VoiceIO.exe** located in:
C:\Amigo\VoiceIO\VoiceIO\bin\Release

A screenshot of a Windows command prompt window titled "C:\Amigo\VoiceIO\VoiceIO\bin\Release\VoiceIO.exe". The text inside the window reads: "VoiceIO Service -- LogicDIS S.A. 2006", "Publishing VoiceIO service...", "The VoiceIO Web Service is now running and ready to be discovered (also WSDL) as", "http://velerefontis:7777/VoiceIO/VoiceIO.asmx", "Press ENTER to close application.", and a single hyphen "-" on the next line.

```
C:\Amigo\VoiceIO\VoiceIO\bin\Release\VoiceIO.exe
VoiceIO Service -- LogicDIS S.A. 2006

Publishing VoiceIO service...
The VoiceIO Web Service is now running and ready to be discovered (also WSDL) as
http://velerefontis:7777/VoiceIO/VoiceIO.asmx
Press ENTER to close application.
-
```

On successful execution the service will be published and ready to accept requests for both recognition and synthesis tasks.

Step2

Launch the application that intends to use the VoiceIO functionality. For demo purposes run the **Client.exe** located in **C:\Amigo\VoiceIO\Client\bin\Release**

The application tries to discover the VoiceIO service and use it...

A screenshot of a Windows command prompt window titled "C:\WINNT\system32\cmd.exe - Client". The text inside the window reads: "VoiceIO Demo -- LogicDIS S.A. 2006", "Discovering VoiceIO service...", and "Found 1 services, trying the first one...".

```
C:\WINNT\system32\cmd.exe - Client
VoiceIO Demo -- LogicDIS S.A. 2006

Discovering VoiceIO service...
Found 1 services, trying the first one...
```

As soon as the application finds the VoiceIO service it synthesizes and plays the greeting and the instructions to the user (*"Welcome to the Voice IO test!" "Please say a four digit number!"*). The wording of these prompts are defined in the corresponding files:
C:\Amigo\VoiceIO\SystemPrompts\greetings.txt and
C:\Amigo\VoiceIO\SystemPrompts\instructions.txt. You can change the phrases the application is going to prompt freely.

A screenshot of a black command prompt window showing two lines of text: "Greeting user..." and "Explain what to do...".

```
Greeting user...
Explain what to do...
```

Then the application starts “listening” for user’s input. The record process is called for a predefined time period.

```
Rec START CALLED
Rec Restarted
rec: 16384
rec: 16384
rec: 16384
rec: 16384
```

The collected user input is transformed to a valid ASR wave format and sent to the speech recognition engine for recognition and understanding against a valid grammar (C:\Amigo\VoiceIO\ApplicationResources\digits4.grxml)

The full recognition result XML as well as the extracted confidence, understood value and literal are displayed in the applications screen.

```
Recognize and understand speech...
Result from recognizer call:
<?xml version='1.0'?><result><interpretation grammar="c:\Amigo\VoiceIO\ApplicationResources\digits4.grxml" confidence="1"><input mode="speech">one three four five</input><instance><anyvar confidence="1">1345</anyvar><SWI_literal>one three four five</SWI_literal><SWI_grammarName>c:\Amigo\VoiceIO\ApplicationResources\digits4.grxml</SWI_grammarName><SWI_meaning><anyvar:1345></SWI_meaning></instance></interpretation><interpretation grammar="c:\Amigo\VoiceIO\ApplicationResources\digits4.grxml" confidence="1"><input mode="speech">one eight four five</input><instance><anyvar confidence="1">1845</anyvar><SWI_literal>one eight four five</SWI_literal><SWI_grammarName>c:\Amigo\VoiceIO\ApplicationResources\digits4.grxml</SWI_grammarName><SWI_meaning><anyvar:1845></SWI_meaning></instance></interpretation></result>

The confidence: 1
The value: 1345
The SWI_literal: one three four five
```

The understood literal is then passed as input to the synthesis engine and the application prompts the user with what it understood that user said.

```
The SWI_literal: one three four five
Say what was understood...

Do you want to repeat? [Y/N] -> _
```

At the end the application asks if the user wants to repeat the entire process or quit.

1.4.2.6 Run VoicelO Server and GenSynthClient Application

Introduction

GenSynthClient application is developed to demonstrate the use of Response Generation module of VoicelO service in combination with the Speech Synthesis and Play methods.

Speech generation systems provide computers with the ability to generate natural language sentences, which then can be synthesized and prompted to user. Current technology does not yet support *unconstrained* speech generation: the ability to generate any sentence in any context and prompt it accurately. To achieve reasonable generation accuracy and response time, current approach constrains what the system generates by using *templates* for describing the possible combinations of words and phrases used to form a response for a given situation or application status.

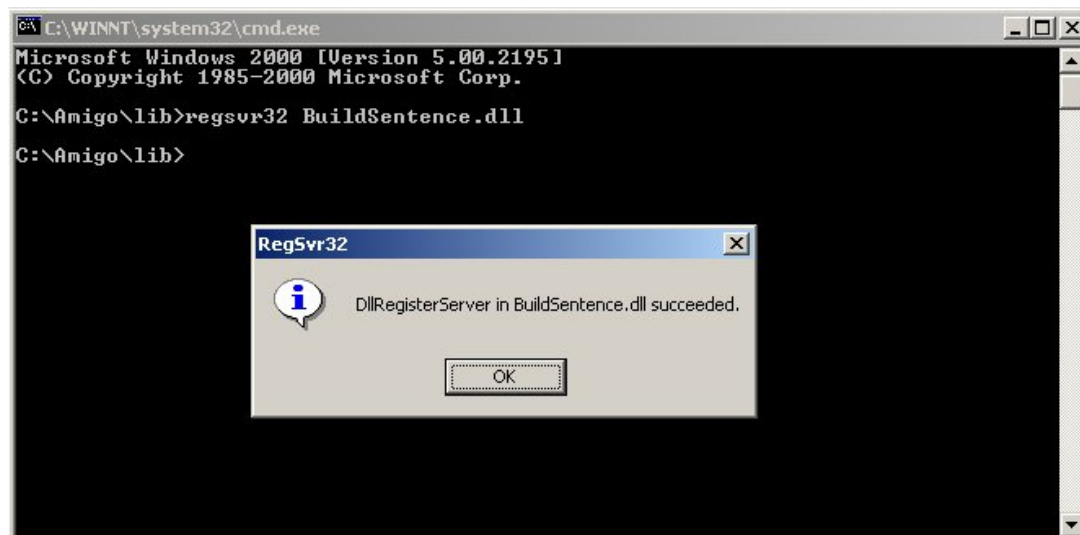
The *Natural Language Generation Template Format* (NLGTF) defines a platform-independent, vendor-independent way of describing one type of template, a *rule based template* similar to the rule based grammars used for speech recognition and understanding. It uses a textual representation that is readable and editable by both developers and computers.

Testing GenSyntClient application

Step 1

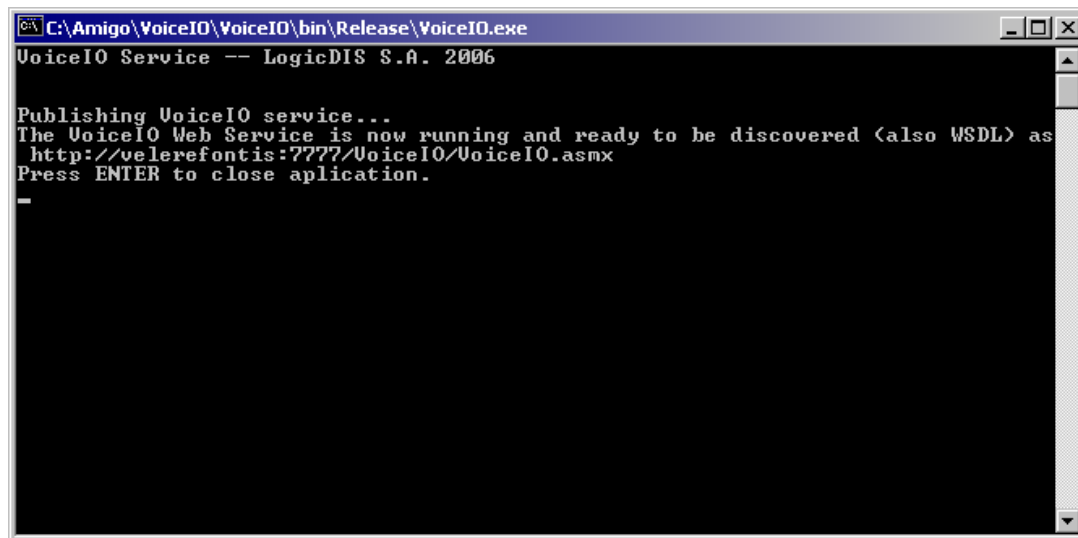
Register the *BuildSentence.dll* library, located in C:\Amigo\lib. From command line enter the following command:

```
>regsvr32 BuildSentence.dll
```



Step 2

Launch the VoiceIO service by executing the **VoiceIO.exe** located in:
C:\Amigo\VoiceIO\VoiceIO\bin\Release

A screenshot of a Windows command prompt window titled "C:\Amigo\VoiceIO\VoiceIO\bin\Release\VoiceIO.exe". The text inside the window reads: "VoiceIO Service -- LogicDIS S.A. 2006", "Publishing VoiceIO service...", "The VoiceIO Web Service is now running and ready to be discovered (also WSDL) as http://velerefontis:7777/VoiceIO/VoiceIO.asmx", and "Press ENTER to close application." followed by a single hyphen character.

```
C:\Amigo\VoiceIO\VoiceIO\bin\Release\VoiceIO.exe
VoiceIO Service -- LogicDIS S.A. 2006

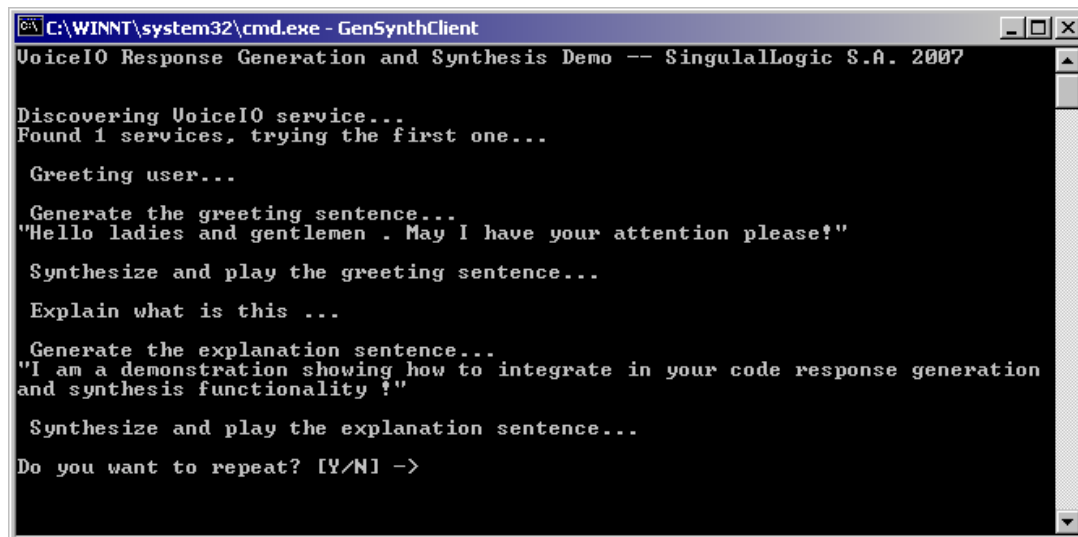
Publishing VoiceIO service...
The VoiceIO Web Service is now running and ready to be discovered (also WSDL) as
http://velerefontis:7777/VoiceIO/VoiceIO.asmx
Press ENTER to close application.
-
```

On successful execution the service will be published and ready to accept requests for both recognition and synthesis tasks.

Step3

Launch the **GenSynthClient.exe** located in **C:\Amigo\VoiceIO\GenSynClient\bin\Release**

The application tries to discover the VoiceIO service and use it...

A screenshot of a Windows command prompt window titled "C:\WINNT\system32\cmd.exe - GenSynthClient". The text inside the window shows the application's startup sequence: "VoiceIO Response Generation and Synthesis Demo -- SingulaLogic S.A. 2007", "Discovering VoiceIO service...", "Found 1 services, trying the first one...", "Greeting user...", "Generate the greeting sentence..." followed by the output "Hello ladies and gentlemen . May I have your attention please!", "Synthesize and play the greeting sentence...", "Explain what is this ...", "Generate the explanation sentence..." followed by the output "I am a demonstration showing how to integrate in your code response generation and synthesis functionality ?", "Synthesize and play the explanation sentence...", and finally "Do you want to repeat? [Y/N] ->".

```
C:\WINNT\system32\cmd.exe - GenSynthClient
VoiceIO Response Generation and Synthesis Demo -- SingulaLogic S.A. 2007

Discovering VoiceIO service...
Found 1 services, trying the first one...

Greeting user...

Generate the greeting sentence...
"Hello ladies and gentlemen . May I have your attention please!"

Synthesize and play the greeting sentence...

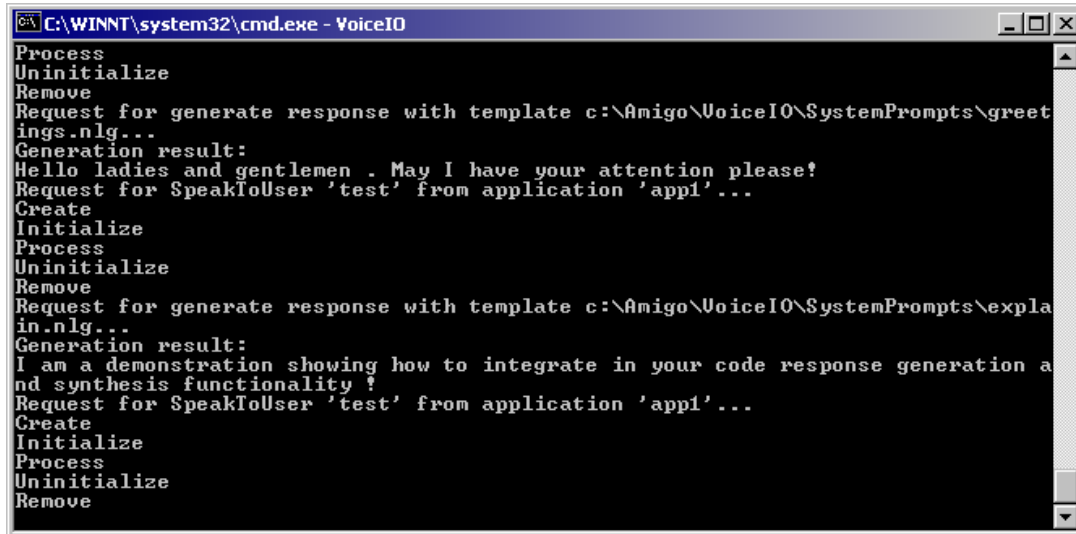
Explain what is this ...

Generate the explanation sentence...
"I am a demonstration showing how to integrate in your code response generation
and synthesis functionality ?"

Synthesize and play the explanation sentence...

Do you want to repeat? [Y/N] ->
```

As soon as the application finds the VoiceIO service it generates, synthesizes and plays the greeting to user (based on greetings.nlg template stored in C:\Amigo\VoiceIO\SystemPrompts\). Then the same procedure is repeated for the explanation (explain.nlg template). Both the client and the server window display the generated response sentence for debugging purposes.



```
C:\WINNT\system32\cmd.exe - VoiceIO
Process
Uninitialize
Remove
Request for generate response with template c:\Amigo\VoiceIO\SystemPrompts\greet
ings.nlg...
Generation result:
Hello ladies and gentlemen . May I have your attention please!
Request for SpeakToUser 'test' from application 'app1'...
Create
Initialize
Process
Uninitialize
Remove
Request for generate response with template c:\Amigo\VoiceIO\SystemPrompts\expla
in.nlg...
Generation result:
I am a demonstration showing how to integrate in your code response generation a
nd synthesis functionality !
Request for SpeakToUser 'test' from application 'app1'...
Create
Initialize
Process
Uninitialize
Remove
```

At the end the application asks if the user wants to repeat the entire process or quit. Valid modifications/transformations can be applied to the nlg templates even on runtime because the generator reloads the current template every time it is called.

If you want to use different template files you should recompile the GenSynthClient project.

2 3D Gesture Service

2.1 Component Overview

2.1.1 Gesture Service

2.1.1.1 3D Gesture Service

Provider

VTT

Introduction

3D gesture service is a part of Amigo User Interface Services. It enables recognition of free form 3D gesture based on wireless handheld sensor device. The sensor device can be used for controlling appliances in three different modes:

1. gesture mode by performing discrete pretrained gesture with the controller device;
2. tilt mode by tilting the controller device;
3. compass mode by rotating the controller device.

The system has optional support for physical selection of objects by means of infrared pointing.

The following guidelines are provided to help to getting started on working with the 3D Gesture Service.

Development status

3D Gesture Service is now available as Amigo Web Service. Gesture trainer works as separate application.

Intended audience

Amigo partners

License

VTT owns the IPR for software and hardware. Licensing is negotiable with separate agreement.

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Hardware:

- VTT SoapBox wireless sensor device with receiver

- SoapBox based Ir tags (optional)
- PC computer with serial port and network connection

Software:

- Windows XP
- LabView runtime engine (supplied with 3D Gesture Service)
- Microsoft .NET Framework v2.0
- Amigo .NET programming framework
- Recogniser.exe
- Trainer.exe

Platform**Tools**

Microsoft Visual Studio 2005

LabView runtime engine (supplied with 3D Gesture Service)

Files

The source codes will be downloaded into the gforge repository under the [amigo] / ius/user_interface / gesture_service / 3D_gesture_service:
The folder 3DGestureService includes .NET based clientserver program.
The folder RecogniserInstaller includes installer software for the recogniser.
The folder TrainerInstaller includes installer software for the trainer.

Documents

Sections 2 and 3 of this document provide information about deployment and architecture of the 3D Gesture Service. Section 4 shortly describes how the 3D Gesture Service and application it uses can be developed.

General description of 3D Gesture Service, its architecture and relation to other Amigo components are given in documents D4.1 and D4.2.

Document 3D_Gesture_Service_software_user_guide provides usage instructions of 3D Gesture Service.

Tasks**Bugs****Patches**

2.2 Deployment

2.2.1 Gesture Service

2.2.1.1 3D Gesture Service

2.2.1.1.1 System requirements

When you deploy 3D Gesture Service simulator or executable, the LabView runtime Engine requires a minimum of

- 64 MB of RAM
- screen resolution of 800×600 pixels
- Pentium 200 MHz or equivalent processor
- 25 MB of disk space

In addition, Amigo .NET programming framework and Microsoft .NET Framework v2.0 are required.

2.2.1.1.2 Download

Files required to run 3D Gesture Service can be downloaded from InriaGforge repository. The address to be used on your subversion client to checkout/commit 3D Gesture Service on the repository is:

```
/svn+ssh://login@scm.gforge.inria.fr/svn/amigo/ius/user_interface/gesture_service/3D_gesture_service/trunk/
```

2.2.1.1.3 Install

3D Gesture Service works by running the GestureService executable. Correspondingly, the example client works by running the GestureClient executable. GestureService uses recogniser application. Alternatively, the service and client applications can be debugged by using the Microsoft Visual 2005 software environment. The service application uses recogniser application. To install the recogniser, complete the following steps:

1. Log on as an administrator or as a user with administrator privileges.
2. Go to the folder RecogniserInstaller \ Volume and double click setup.exe. This should start installation of LabView 8 runtime engine. NOTE: if you have already installed runtime engine for LabView 8 you can go to step 3.
3. Copy the files (created by the installer)

Recogniser.aliases;
Recogniser.exe;
Recogniser.ini;

into the following directories:

```
<ROOT_PATH> \ 3DGestureService \ GestureService \ GestureService \ bin \ debug;  
<ROOT_PATH> \ 3DGestureService \ GestureService \ GestureService \ bin \ release;
```

2.2.1.1.4 Configure

Further configurations depend on the training procedure (i.e. user names, what and how many gestures are used etc.)

2.2.1.1.5 Compile

3D Gesture Service is provided as an executable - no compilation is required.

2.3 Component Architecture

2.3.1 Gesture Service

2.3.1.1 3D Gesture service i

Component interface

3D gesture client subscribes the service by calling the SetParameters(string userID, string protocol) method. UserID refers to the user name used with training procedure and protocol refers to the protocol used to exchange the gesture events. Protocol options are mmil, soap, http and skip (just simple text string). Below are protocol examples of user "Test" performing gesture "Play" by using controller "85" with confidence between 7883..

Example 1. mmil message

```
<mmil:event id="e0">
<mmil:evtType>3D reco</mmil:evtType>
<mmil:dialogueAct>inform</mmil:dialogueAct>
</mmil:event>
<mmil:event id="e1">
<mmil:evtType>report</mmil:evtType>
<mmil:UserID>Test</mmil:UserID>
<mmil:DeviceID>85</mmil:DeviceID>
<mmil:eventType>gesture</mmil:UserType>
<mmil:actionStatus>performed</mmil:actionStatus>
<mmil:mode></mmil:mode>
</mmil:event>
<mmil:participant id="p0">
<mmil:GestureName>Play</mmil:GestureName>
<mmil:Confidence>78</mmil:Confidence>
</mmil:participant>
<mmil:relation laf:source="p0" laf:target="e1"
type="description"/>
```

Example 2. soap message

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
gesture=Play
confidence=79
</soap:Body>
</soap:Envelope>
```

Example 3. http headers

```
POST / HTTP/1.1
gesture=Play
confidence=83
```

Example 4. simple text string

```
gesture=Play
confidence=83
```

The gesture service passes gesture events to the client through WSEventing.
The client subscribes to gesture events by using WSEventing method

Subscribe(string eventName, WSEventingSink.NotificationDelegate notificationDelegate,
DateTime expires).

On the server side, special WSEvent attribute gstNotification is used to export gesture events as remote events through WSEventing to the client.

Mechanisms of interaction

When gesture service is launched it needs to be initialized with the following parameters: hostname of your computer and name of the serial port where the controller (SoapBox) is connected. The service application asks user to this information. After the service is running the client can be launched. After the client has subscribed the service by using SetParameters method, it starts receiving gesture events every time they occur.

The client side does not handle the error situation if the server falls out. So the preferred order to shut the clientserver system down is to (1) exit client, (2) exit service and (3) close the recogniser application which the service uses.

Overview and reference to internals

-

Detailed documentation

D41. and D4.2 provides general descriptions about the architecture and functionalities of the 3D Gesture Service.

2.4 Tutorial

2.4.1 3D gesture Service

Component development

The gesture recogniser and trainer are provided as Windows executables and can not be modified. The clientserver program can be further developed, if needed, by using Microsoft Visual Studio 2005 software environment.

Application development

The clientserver program is based on .NET. Other possibility is to use GstRecogniser.exe which is available at Gforge (see further instruction below, non-web service case). It (client socket) uses socket to send gesture events to application (server socket). In this way, the application can be developed independently on any programming language. Unfortunately, the hardware availability is limited. However, the actual gesture recognizer is independent of the used acceleration sensor hardware, so if in some point there is need to change or update the sensor hardware, the recognition software interface towards the application remains unchanged. During the project, the gesture control hardware can be borrowed from VTT with separate agreement.

3G Gesture recognition service, Non-Web Service case

Environment (set-up) info needed if you want to run this sw (service)

Hardware:

- VTT SoapBox wireless sensor device with receiver
- SoapBox based Ir tags (optional)
- PC computer with serial port and network connection

Software:

- Windows XP
- LabView runtime engine (supplied with 3D Gesture Service)
- 3D Gesture Service software

Platform

Tools

LabView runtime engine (supplied with 3D Gesture Service)

Files

LabView runtime engine

- Folder **Volume**

3D Gesture Service simulator

- Folder **data**
- Files **GstSimulator.aliases**, **GstSimulator.exe**, **GstSimulator.ini**

3D Gesture Service executable

- Folder **data**
- Files **GstRecogniser.aliases**, **GstRecogniser.exe**, **GstRecogniser.ini**

Documents

Sections 2 and 3 of this document provide information about deployment and architecture of the 3D Gesture Service. Short tutorial is given in Section 4. When the full 3D Gesture Service is ready, the document will be updated accordingly.

General description of 3D Gesture Service, its architecture and relation to other Amigo components are given in documents D4.1 and D4.2.

Tasks

Bugs

Patches

2.5 Deployment

2.5.1 Gesture Service

2.5.1.1 3D Gesture Service

2.5.1.1.1 System requirements

When you deploy 3D Gesture Service simulator or executable, the LabView runtime Engine requires a minimum of

- 64 MB of RAM
- screen resolution of 800×600 pixels
- Pentium 200 MHz or equivalent processor
- 25 MB of disk space

2.5.1.1.2 Download

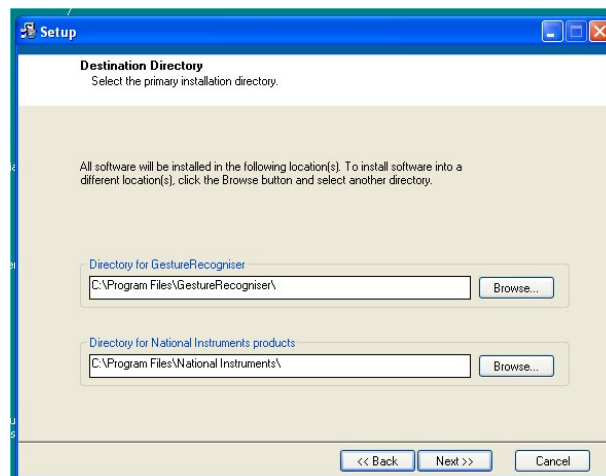
Files required to run 3D Gesture Service can be downloaded from InriaGforge repository. The address to be used on your subversion client to checkout/commit 3D Gesture Service on the repository is:

```
/svn+ssh://login@scm.gforge.inria.fr/svn/amigo/ius/user_interface/gesture_service/3D_gesture_service/trunk/
```

2.5.1.1.3 Install

Complete the following steps to install 3D Gesture Service.

1. Log on as an administrator or as a user with administrator privileges.
2. From the downloaded 3D Gesture Service files, go to the folder **Volume** and double click **setup.exe**. This should start installation of LabView 8 runtime engine. NOTE: if you have already installed runtime engine for LabView 8 you can go to step 3.



After installation of LabView Runtime Engine has completed, you should restart your computer.

3. From downloaded 3D Gesture Service files, copy folder **data** and files **GstSimulator.aliases**, **GstSimulator.exe** and **GstSimulator.ini** to your preferred location.
4. a) To run 3D Gesture Service simulator, double click **GstSimulator.exe**.
b) To run 3D Gesture Service executable, double click **GstSimulator.exe**.

2.5.1.1.4 Configure

3D Gesture Service is provided as an executable - no configuration is required

2.5.1.1.5 Compile

3D Gesture Service is provided as an executable - no compilation is required.

2.6 Tutorial

2.6.1 3D gesture Service

2.6.1.1 Component development

The 3D gesture recognition component is provided as Windows executable and it is not possible to modify it.

2.6.1.2 Application development

Applications that utilize 3D gestures can be developed independently from the gesture recognition hardware using provided gesture simulator software. Since the hardware availability is limited it is recommended to start the application development using the simulator software. The actual gesture recognizer is independent of the used acceleration sensor hardware, so if in some point there is need to change or update the sensor hardware the recognition software interface towards the application remains unchanged. During the project gesture control hardware can be borrowed from VTT with separate agreement.

Gesture recognition output XML formatting is user selectable. It can be either simple text protocol or INRIA's MMIL XML format and users can select if to use SOAP message headers. The application has to provide TCP/IP server for communication and appropriate parsers for the XML messages. Later the 3D gesture recognition service interface will be changed to Web Service interface.

Example below shows message for gesture "Up" using simple text protocol and MMIL:

Simple text protocol:

gesture = Up

confidence = 65

MMIL:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
```

```
  <soap:Body>
```

```
    <mmil:event id="e0">
```

```
      <mmil:evtType>3D reco</mmil:evtType>
```

```
      <mmil:dialogueAct>inform</mmil:dialogueAct>
```

```
    </mmil:event>
```

```
    <mmil:event id="e1">
```

```
      <mmil:evtType>report</mmil:evtType>
```

```
      <mmil:UserID>Juha</mmil:UserID>
```

```
      <mmil:DeviceID>20</mmil:DeviceID>
```

```
      <mmil:eventType>gesture</mmil:UserType>
```

```
      <mmil:actionStatus>performed</mmil:actionStatus>
```

```
<mmil:mode></mmil:mode>
</mmil:event>

<mmil:participant id="p0">
  <mmil:GestureName>Up</mmil:GestureName>
  <mmil:Confidence>65</mmil:Confidence>
</mmil:participant>

<mmil:relation laf:source="p0" laf:target="e1" type="description"/>

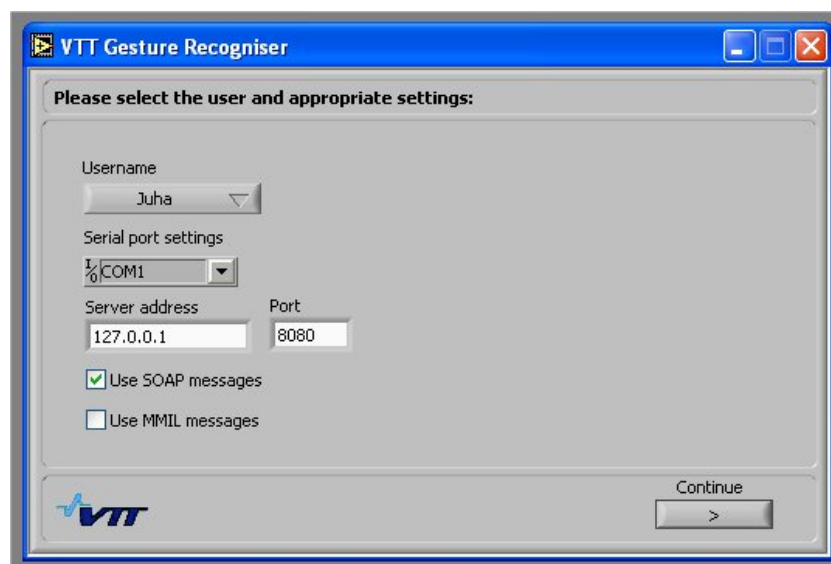
</soap:Body>
</soap:Envelope>
```

In this example user “Juha” has performed gesture “Up” using gesture controller “20” and the confidence of the gesture was 65% (gesture quality). Gesture simulator software can be used to generate gesture events “Up”, “Down”, “Left” and “Right”.

Gesture Simulator and executable usage instructions:

Gesture recogniser executable works as follows:

1. Start **GstSimulator.exe** or **GstRecogniser.exe**.
2. Select the username by double clicking it from the list and set the other parameters required by your application. Then press **Continue**.



3. Gesture mode works by pressing the controller's button 1 (see figure below) i.e., you press the button down in the beginning of the gesture and release it when the gesture

is finished. Button works as a “gesture capture” button i.e. the button must be pressed for the whole duration of the gesture.of the controller. The recognised gesture event appears in **Recognised gesture** field and **command sent** window shows the message sent according your settings.

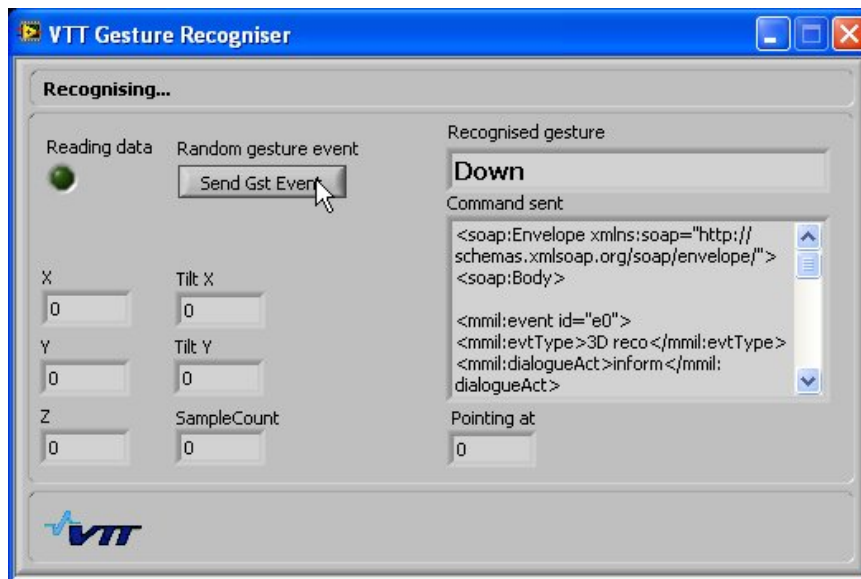


4. Tilting mode works when you press the button 2 (see figure below) of the controller. Again, the recognised tilt event appears in **Recognised gesture** field and **command sent** window shows the message sent according your settings.



5. The recogniser application can be closed by pressing the Exit button.

Figure below presents the user interface of the Gesture simulator. Instead of using specific control HW random gesture events are generated by button (Send GST Event) in simulator UI.



3 Multi-modal Interface Services / Multi-device and Dialog Management Service

3.1 Component Overview

3.1.1 Voice Service

3.1.1.1 Implicit Speech Input

Provider

INRIA

Introduction

In the first part of the project, the aim of this task was to design a generic architecture to help application developers to exploit users' implicit speech input. In the second phase of the project, the objective of this task has been to focus on one particular type of implicit speech information, and to provide it to Amigo application developers. Two kinds of implicit speech information have been first considered: automatic dialog act recognition, and topic recognition. After some research work on both aspects, INRIA has decided to choose and develop **automatic topic recognition** in the Amigo project, while dialog act recognition will be studied mainly as paperwork and will not be implemented in the context of Amigo.

Automatic topic recognition is a particular kind of implicit speech interaction, because it transparently – without disturbing the user – exploits user's speech. More precisely, it is **implicit**, because the user's speech is not originally intended to communicate with the system, but rather to communicate with another human. Typically, the automatic topic recognition functionality might infer the current topic of discussion from two people talking together face-to-face, or from two people talking on the phone.

One of the main requirements of topic recognition is its low memory and computational requirements: indeed, such an implicit system is designed to run everywhere, permanently and for many – if not all – users. This is hardly achievable when it requires a lot of resources. This is why we have quickly given up the first option, which was to connect the output of a state-of-the-art large vocabulary automatic speech recognition system to the input of our topic recognizer. We have rather decided to investigate and design a lightweight spoken keywords recognition system instead, which is dedicated to work as a pre-processor to the topic recognition module. The efforts concerning the topic recognition module have thus been distributed into both task 4.1 “CMS topic recognition” and subtask 4.5.1 “implicit speech input” as follows:

- Task 4.1 deals with the design and development of the inference engine that recognizes topic from a text stream or a sequence of words. It also deals with making the topic recognition fully compliant with the context management system, in particular implementing the *IContextSource* interface, supporting SPARQL queries and RDF descriptions, and interacting with the context ontology.
- Subtask 4.5.1 deals with developing a lightweight keyword spotting module, which can of course be used as a standalone module, but that is primarily designed to extract the most important keywords that can then be passed to the topic recognition module from the user's speech.

Development status

The second version of the keyword spotting module is available: it supports speaker-dependent real-time keywords recognition, and it is fully integrated into the topic recognition OSGI context source Amigo component.

Intended audience

Project partners

Developers**License**

LGPL

Language

Java and C

Environment (set-up) info needed if you want to run this sw (service)

Windows XP, cygwin, JDK1.5, OSCAR, Amigo CMS bundles

Platform

JVM, OSGi-Based Programming & Deployment Framework from WP3

Tools

Oscar

Files

Available for download at

[amigo_gforge]/ius/user_interface/voice_service/implicit_speech_input/

Documents

The usage and deployment of the keyword recognition module are fully described in the topic recognition user and developer guide, which can be found on the Amigo Gforge repository.

Tasks**Bugs**

Patches**3.1.2 Gesture Service****3.1.2.1 2D Gesture Service****Provider**

INRIA/LORIA

Introduction

The package comprises two services: a service which displays a scene on a tactile screen and captures any gesture drawn on it with a special pencil (GestureCapture) and a service which interprets the gesture (GestureInterpreter). The two services work with MultiModal Interface Language representations (MMIL). The scene, encoded in MMIL, is a set of objects with different states represented visually by images and masks. Any gesture drawn on the screen could lead to a MMIL event sent by the GestureCapture service to the GestureInterpreter service. The content of the event is the set of all the selected objects in the scene, associated to salience measures. The GestureCapture service also handles ways to modify the scene by adding objects, changing their aspect, or their states.

Development status

Close to final version

Intended audience

Project partners

Developers

Alexandre Denis – alexandre.denis@loria.fr

License

LGPL

Language

Java 1.5

Environment (set-up) info needed if you want to run this sw (service)

Any OS with JRE1.5

Platform

JVM, OSGi-Based Programming & Deployment Framework from WP3

Tools

Oscar

Files

Available for download at

[amigo_gforge]/ius/user_interface/gesture_service/2D_gesture_service/

Documents

Design can be found in D4.1 and D4.2, Designers guide is this document.

Tasks

None

Bugs

Selection problem in particular avoidance gestures

Patches**3.1.3 Dialogue Manager****Provider**

Design: INRIA and all partners, implementation: SIT

Introduction

The dialogue manager shall facilitate the work of the application developer by proposing common mechanisms to access and control the fundamental user interface services (e.g. automatic speech recognition, gesture recognition, GUI) as well as the more advanced user interface services (e.g. multi-device service, context-driven user interactions).

In the context of ambient intelligence, it shall further provide and use contextual information that could be used, for example to trigger new applications. Hence, both explicit and a number of implicit interactions can be handled by the dialogue manager.

As the precise functionalities of future interaction services can not be known in advance, the architecture of the Dialogue Manager shall also be very flexible so that these new future services can be easily integrated. Therefore, we have chosen to base the Dialogue Manager on the blackboard design pattern, as described in the following sections.

The PEGASUS framework enables applications to interact using the blackboard design pattern. This framework provides the basis for the Amigo service Dialogue Manager that allows arbitrary Amigo components to interact with Amigo's various user interface services.

Development status

Architectural design has been completed. An instance of the Dialogue Manager, with a limited set of functionalities, has been implemented and integrated in the WP6 “Role-Playing Game” demonstrator. The Amigo service, which implements the Dialogue Manager, is still under development. The PEGASUS framework is finished.

Intended audience

Project partners

Developers

Design: INRIA and all WP4 partners

Implementation: SIT

License

Application dependent

Language

Java and C/C++

Environment (set-up) info needed if you want to run this sw (service)

Any Windows with JRE1.5

Platform

JVM, OSGi-Based Programming

Tools

OSCAR

Files

No source files available yet

Documents

Designers guide is this document.

A detailed documentation of the PEGASUS framework is available at the gforge repository.

Further information is available in the form of internal WP4 working documents.

Tasks

None

Bugs

None

Patches

None

3.1.4 Multimodal Fusion**Provider**

INRIA

Introduction

The Multimodal Fusion Module (MFM) deals with related information from the user, occurring in short periods of time and that can be mutually completed. The role of the MFM is to merge and cross-complete information from several input modules. The MFM then outputs the merged information with as primary target the Multimodal Dialog Manager.

- Inputs: several MMIL-encoded message. MMIL format is used for each of the input modalities, a conversion module may be required for each modality that is not handled by a MMIL-compliant module.
- Output: a unique MMIL message containing all the merged messages, completed with complementary information when possible.
- Parameters:

An ontology,

Descriptions of the currently accessible task entities and events/commands

A modality-dependant tuning about time windows is possible, which can be turned, if necessary, into a context-dependant tuning, while it is not the case currently.

Currently handled modalities: Natural Language, 2D Gestures (object selections), 3D Gestures (commands)

- Semantic chunks (from natural language analysis) with a primary type (e.g. from the noun for participants or verbs for events: the TV, stop) and/or a set of atomic characteristics (e.g. from adjectives or verbs). Complex characteristics related to events or related to states with one relations such as “the pen I left on the chair yesterday” or “the pen near the box” are not handled in the current state of development.
- Weighted sets of conceptually-marked objects (from 2D gesture modules)
- Commands related to the task (from 3D gesture modules, menu-oriented UI)
- Following examples are all eventually merged into a simple command Pause(HiFi):
- User gesture describing a pause THEN user gesture pointing toward the HiFi.
- User utterance “The first-floor stereo” WHILE user gesture describing a pause.
- User utterance “Stop the music”.

Development status

Stable. The MMF is implemented as an Amigo service.

Intended audience

Project partners

Developers

Guillaume Pitel

License

LGPL

Language

Java 1.5

Environment (set-up) info needed if you want to run this sw (service)

Any OS with JRE1.5

Platform

JVM, OSGi-Based Programming & Deployment Framework from WP3

Tools

Oscar

Files

Available for download at [\[amigo_gforge\]/ius/user_interface/multimodal_fusion/](#)

Documents

Design can be found in D4.1 and D4.2, Designers guide is this document.

Tasks**Bugs****Patches**

3.2 Deployment

3.2.1 Implicit Speech Input

3.2.1.1 System requirements

Amigo OSGI framework (amigo_core, amigo_ksoap, amigo_wsdiscovery)

OSGI framework

OSCAR platform (can be on either linux or windows)

Refer to [OSCAR req]

3.2.1.2 Download

Source code available for download at

[amigo_gforge]/ius/user_interface/voice_service/implicit_speech_input/

3.2.1.3 Install

The keyword spotting module is integrated into the topic recognition bundle: you need to download this bundle and install it as any other bundle into OSCAR. Then, you need to extract and uncompress a zipfile that is stored in this bundle, and run a cygwin script file, which will launch the keyword spotting software. The precise procedure is described in the user and developer guide of the topic recognition module.

3.2.1.4 Configure

The keywords spotting module reads in a configuration file that defines the list of supported keywords. The keyword spotting module further needs a pronunciation lexicon for these keywords, as well as a set of speaker-dependent acoustic models. This procedure is described in details in the user and developer guide of the topic recognition module.

3.2.1.5 Compile

Two different source file packages are used for implicit speech: a “light” package, that only contains the JAVA sources and that is included into the topic recognition module. This light version handles basic keyword spotting and is fully integrated within the topic recognizer. You should compile it using the ant build script of the topic recognition module.

The second package contains the full set of source codes, including the Julius C speech decoder, along with several toolkits to train adapted acoustic models: this set is stored in the **implicit speech** repository, and is independent from the topic recognizer. To compile it, you must first compile the Julius decoder (in cygwin) using the ./configure and ./make procedure. Note that this version of the decoder has been modified for the need of the project: a blank/original Julius package will not work. Then, you can use the ant build script to compile the JAVA part.

3.2.2 2D Gesture Service

3.2.2.1 System requirements

OSCAR platform (can be on either linux or windows)

OSGI framework

The following AMIGO OSGI services should be running from the OBR before starting 2D services :

- amigo_core
- amigo_ksoap_binding
- amigo_ksoap_export
- amigo_wsdiscovery
- HTTP Service (+ Amigo mods)
- Servlet
- Service binder
- log4

3.2.2.2 Download

Source code available for download.

3.2.2.3 Install

The gesture capture and interpreter services are located in /amigo/ius/user_interface/gesture_service/2D_gesture_service/. Compile the two services, and copy the bundles located in dist directory to \$OSCAR_HOME/bundle.

The services should be started in the following order : GestureCapture then GestureInterpreter. That is because the GestureInterpreter subscribes to the gesture events sent by the first GestureCapture it finds with the lookup.

In Oscar :

```
start file:bundle/gesture-capture.jar
```

```
start file:bundle/gesture-interpreter.jar
```

3.2.2.4 Configure

Configuring the services

Once the services are running, the scene should be configured by calling the *setScene* method of both services using a stub. **Warning** : don't be confused, if a scene is displayed on GestureCapture frame, it does not imply that the GestureInterpreter is configured also. One should configure it by calling the *setScene* method in order to interpret the gesture. The scene could be retrieved from a file or directly from a MMILEntity.

An additional service called SceneConfigurator has been provided in order to show how the scene could be configured on both services. It is located in the same directory than the interpreter and capture services. The service is an OSGI service, and is launched with (in Oscar) :

```
start file:bundle/scene-configurator.jar
```

Once it is started, you just have to push the configure button and choose a valid scene file. The configurator will call the *setScene* method of interpreter and capture services.

Scene description

The scene has been improved compared to previous version. Now a scene is a set of objects, each one described by a set of states represented by sprites and masks. Each state is associated to an identifier which allows changing the current state of an object dynamically. In addition, the polygons now serve as masks to identify selectable areas in an object.

The scene is encoded in the MMIL format by a participant whose objType is Scene. The objects of the scene are represented as sub-participants which should have at least :

- a *mmil:mmilId* feature which is a unique id identifying the object in the amigo framework (the id attribute of the participants identifies them only in the component),
- a *vis:position* feature which represent its coordinates in the scene,
- at least one sprite.

A sprite is defined as objects' sub-participants, they are defined by :

- a *mmil:objType* feature which has always "Sprite" for value
- a *mmil:mmilId* feature identifying uniquely the sprite
- an optional *vis:image* feature giving the url of the image
- an optional mask encoded by a *vis:maskPoints* feature
- aspect features (fill, visible, color)

A sprite should have a *vis:image* or a *vis:maskPoints* to be represented. The following table explains the look of an object according to the presence or absence of these features.

| Look of an object | <i>vis:image</i> present | <i>vis:image</i> absent |
|--------------------------------------|--|--|
| <i>vis:maskPoints</i> present | the object is an image with the selectable areas given by the mask | the object is a polygon with no background image |
| <i>vis:maskPoints</i> absent | the object is an image fully selectable (a mask is created) | an exception is thrown while building the scene |

List of visual features:

The visual namespace encodes all the visual aspects of the scene. In the following list :

- a point is formatted as <x> <comma> <y> and a list of points are points separated from each other by spaces (x, y integers).
- a color is either a symbolic value in { black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow } or a numeric rgb triplet <red> <comma> <green> <comma> <blue> <comma> <alpha>, where <red>, <green>, <blue> and <alpha> are integers ranging from 0 to 255.

vis:image = optional URL of a background image

vis:position = a position in the scene (point)

vis:zIndex = the z-index representing the depth of the object, the lowest z-index corresponds to the most distant object (integer)

vis:x = x coordinate (integer)

vis:y = y coordinate (integer)

vis:maskPoints = list of the points of the mask of a sprite (list of points)

vis:maskColor = color of a mask

vis:maskFilled = if the mask is a surface or a contour (true or false)

vis:maskVisible = if the mask should be drawn or not (true or false)

Here a scene example with no background image which contains only one object with two states :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<mmil:mmilComponent xmlns:mmil="mmil.org">
```

```
  <mmil:participant xmlns:mmil="mmil.org" xmlns:vis="vis.org">
```

```
    <mmil:mmilId>officeScene</mmil:mmilId>
```

```
    <mmil:objType>Scene</mmil:objType>
```

```
    <mmil:participant id="laptop">
```

```
      <mmil:objType>Laptop</mmil:objType>
```

```
      <mmil:mmilId>laptop</mmil:mmilId>
```

```
      <vis:position>100,100</vis:position>
```

```
      <mmil:participant id="laptop_spr1">
```

```
        <mmil:objType>Sprite</mmil:objType>
```

```
        <mmil:mmilId>open</mmil:mmilId>
```

```
        <vis:image>file:etc/office/laptop_open.gif</vis:image>
```

```
        <vis:maskFilled>true</vis:maskFilled>
```

```
        <vis:maskPoints>20,214 23,172 19,26 24,23 245,17 248,22 262,165 311,205 311,216 306,220 26,228
21,224</vis:maskPoints>
```

```
      </mmil:participant>
```

```
      <mmil:participant id="laptop_spr2">
```

```
        <mmil:objType>Sprite</mmil:objType>
```

```
        <mmil:mmilId>closed</mmil:mmilId>
```

```
        <vis:image>file:etc/office/laptop_closed.gif</vis:image>
```

```
        <vis:maskFilled>true</vis:maskFilled>
```

```
        <vis:maskPoints>20,206 23,169 28,166 259,159 309,196 310,204 311,216 304,221 27,230
20,225</vis:maskPoints>
```

```
      </mmil:participant>
```

</mmil:participant>

</mmil:participant>

</mmil:mmilComponent>

3.2.2.5 Compile

You can use directly ant on the build.xml file of each subproject, or the build.xml file at the root of the 2D_gesture_services directory which compiles everything.

3.2.3 Multimodal Dialogue Manager

3.2.3.1 System requirements

Amigo OSGI framework (amigo_core, amigo_ksoap, amigo_wsdiscovery)

Amigo CMS service

OSGI framework

OSCAR platform (on Windows)

3.2.3.2 Download

Not available for download yet

3.2.3.3 Install

The Dialogue Manager is comprised of two parts: The Amigo service that is written in Java and set of precompiled windows binaries (PEGASUS framework) supplied in an archive file, which implement the black board pattern. The windows binaries need to be extracted on host and then started using the start script (start_server.bat) that is also provided in the archive. The Amigo service is provided in a bundle that needs to be installed and setup just as any other bundle in OSCAR.

3.2.3.4 Configure

The Amigo service and the PEGASUS server are delivered preconfigured. A detailed description of the configuration and setup possibilities of the PEGASUS framework are supplied in an extra document, available at gforge (Pegasus-Manual.pdf). This document also contains the developer documentation for PEGASUS.

3.2.3.5 Compile

The detailed compilation instructions for the PEGASUS framework are supplied in the extra document "Pegasus-Manual.pdf". To compile PEGASUS, Microsoft Visual Studio 2003 is recommended, the source code of PEGASUS is supplied as a MS Visual Studio 2003 project. To compile the Amigo service, it is only necessary to run the ant script "build.xml", which is bundled with the source code of the service.

3.2.4 Multimodal Fusion

3.2.4.1 System requirements

Amigo OSGI framework (amigo_core, amigo_ksoap, amigo_wsdiscovery)

OSGI framework

OSCAR platform (can be on either linux or windows)

Refer to [OSCAR req]

3.2.4.2 Download

Source code available for download

3.2.4.3 Install

In the repository, two projects are stored under `ius/user_interface/multimodal_fusion/trunk/subproject:` `multimodal_fusion` and `multimodal_fusion_tester`. Both are regular AMIGO services, launched via Oscar (or any OSGi implementation - the tester requires a Shell service), and require the `amigo_ksoap_binding`, `amigo_ksoap_export`, `amigo_wsdiscovery` and `amigo_meshslp_lookup` services.

The bundles are stored in `multimodal_fusion/build/dist/multimodal_fusion.jar` and `multimodal_fusion_tester/build/dist/multimodal_fusion_tester.jar` after compilation (both projects respect the guidelines from the Amigo skeletons projects).

3.2.4.4 Configure

Necessary libraries are contained in the bundles, so that no configuration is required on the classpath. A configuration file for the main bundle is found in `res/mmf.properties`, containing time window definitions for each modality (in milliseconds) – see appendix. No other static configuration is necessary before launching the main service (launching by itself is done the normal way under Oscar: `install <url of the jar>`).

3.2.4.5 Compile

Using eclipse [.project] you can compile a new version of the service (and the test service as well). You can also use directly ant on the `build.xml` file.

3.3 Component Architecture

3.3.1 Implicit speech input component interface

The keyword spotting module is primarily designed to be used as a front-end to the topic recognition Amigo context source.

Nevertheless, if you really want to use it as a stand-alone module, please use the following procedure:

1. First, you need to instantiate a JAVA **JulModule** object, which represents your single access point to the keyword recognizer. The JulModule object actually communicates via socket with the real keyword recognizer, which is written in C. The JulModule object further implements the JAVA swing **Box** interface that shows the keyword spotting controls, which makes it easy for your application to add it to any of your graphical user interface.
2. Then, you need to implement the JAVA **RecoSubscriber** interface, and to subscribe to the JulModule object by calling the *JulModule.subscribe(RecoSubscriber)* method. Whenever a new sentence has been recognized by the keyword spotting module, the method *RecoSubscriber.pushText(String)* is called.

The interfaces are summarized next:

```
public interface fr.loria.parole.topicRecognition.RecoSubscriber
```

This interface shall be implemented by any object that exploits the output of the keyword recognition module.

Method Summary

| | |
|------|---|
| void | pushText (java.lang.String s) This method is called by the keyword recognition module whenever a new sentence has been recognized from the speaker. |
|------|---|

```
public class fr.loria.parole.topicRecognition.JulModule extends javax.swing.Box
```

This class is the main access point to the keyword recognition module. It shall be instantiated by the application that requires access to the keyword recognition module. This class further implements the javax.swing.Box interface, which makes it easy to include into any of the application GUI. It then displays the GUI controls of the keyword recognition module.

Method Summary

| | |
|------|--|
| void | subscribe (RecoSubscriber sub) This method MUST be called by the application that needs to be warned when the user has uttered a new sentence. |
|------|--|

3.3.2 Gesture Service

3.3.2.1 2D Gesture service interfaces

3.3.2.1.1 GestureCaptureService

The javadoc for the GestureCaptureService interface is:

| Method Summary | |
|-----------------------|--|
| void | gestureComplete (MMILGesture gesture) Method executed when a gesture is complete. |
| MMILScene | getScene () Return the current scene. |
| void | setScene (MMILScene scene) Set the current scene. |
| void | setScene (String scene) Convenient method to set the current scene. |
| void | changeState (MMILPercept percept, String state) Method executed when the state of an object changes |

3.3.2.2 GestureInterpreterService

| Method Summary | |
|-----------------------|--|
| MMILScene | getScene () Get the current scene. |
| MMILComponent | interpret (MMILGesture gesture) Interpret a gesture in the current scene. |
| void | setScene (MMILScene scene) Set the scene for current resolutions. |
| void | setScene (String scene) Convenient method to set the scene for current resolutions. |
| void | changeState (MMILPercept percept, String state) Method executed when the state of an object changes |

3.3.3 Multimodal Dialogue Manager

3.3.3.1 Limitations and scope of services

The UIS Dialogue Manager (DM) is a blackboard based system i.e. shared memory (database) area where applications and modalities, i.e. DM clients, can share their inputs and outputs. DM manages the communication between modality services (speech, GUI gestures...), Multi Modal Fusion (MMF) and applications. The architecture is flexible making it possible to add new modalities, applications and data abstraction modules (such as MMF and application specific dialogue managers) as the system grow.

Figure 1 presents the conceptual architecture of the UIS Dialogue Manager. Applications, modality services and MMF first register themselves as UI event producers and consumers. When new applications are registered to DM they have to provide their user interface XML description (See GUI service ServiceDescription parameter format), which describes the functions and capabilities of the application. DM will pass the description to XUI service which generates the optimised and personalised UI representation that can be used in automatic generation of GUI and speech UI representations. Since UI generation is based on same user and environment models the presented output (GUI or voice) will have uniform look and feel. Application specific task and dialogue management is applications responsibility. DM provides flexible mechanism for UI event sharing and synchronisation, but it cannot provide application specific task management.

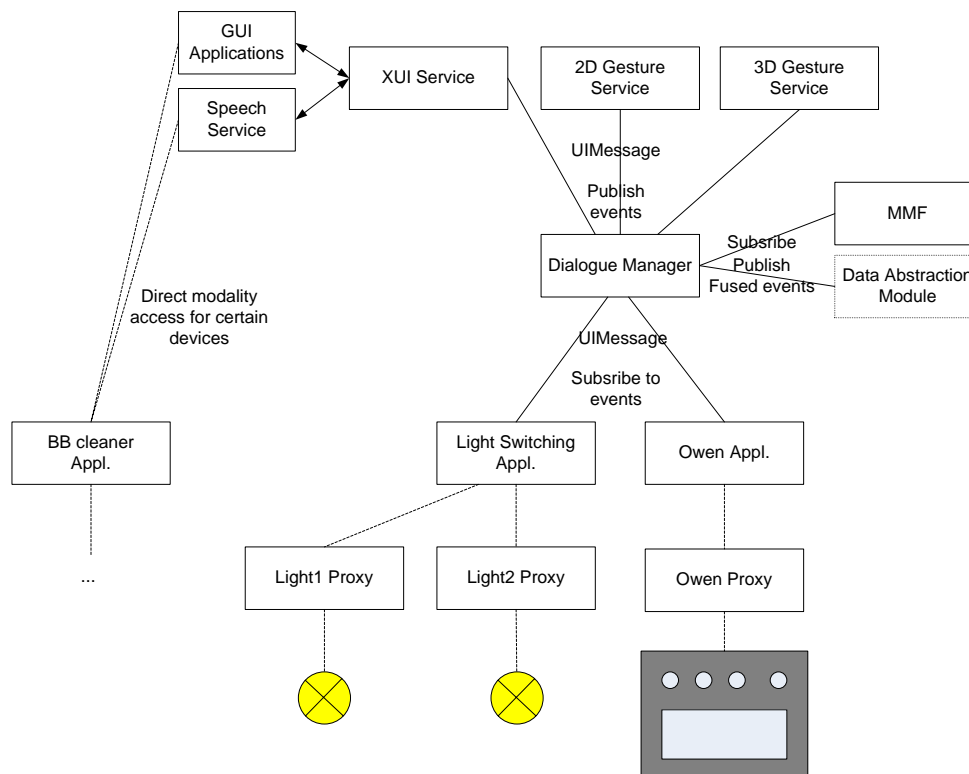


Figure 1. Conceptual architecture of the UIS.

There are different reasons, why a user interacts with the Amigo System. Amigo UIS should support the following interaction modes:

- **Operation mode:** The interaction has been initiated by the user. He wants to change something. So the goal of the interaction has to be told to the system or has to be discovered by the system. Using a GUI or speech the user has to select the operation out of the offered functionalities of the whole services. He has also to specify values needed for the responsible service. Because there could be more than one hundred functionalities offered by the services of one home, these functions have to be sorted into groups and an arbitrary number of subgroups to give the user a structured overview. The functions and its groups could then be represented for example by a menu hierarchy either by GUI or speech. The operation mode includes also the possibility to check the status of different services and the values of parameters of this service. To be used by the operation mode a service has to describe itself, to define possible user interaction functionalities and to give a model of its internal structure to the UIS. The UIS will use data models, dependency-graphs and user dependent menu logic descriptions to build a personalised menu structure.
- **Event mode:** The event mode offers the system the possibility to initiate an interaction with the user. The dialog flow and the requested input parameters are fully controlled by the application interacting with the user. The event mode can be used for warnings, messages, questions etc.

3.3.3.2 Integration with the Context Management Service

User interactions form relevant context information, especially when considering implicit user interactions. For instance, an application can react to specific contextual information, including user location, but also user gestures and user speech. Two options may thus exist: the first one is that every user interaction service implements the Context Source interface. We have not opted for this solution, as this may impact the performances of the overall UIS. Furthermore, it dramatically increases the variety of context sources related to user interfaces, and context consumers may have difficulties to choose the most relevant one.

The second option consists in providing only one context source, namely the dialogue manager. Then, every user interface provider and consumer interacts with the dialogue manager, which centralizes all the contextual information related to user interaction.

A typical usage scenario can take the form (Figure 2):

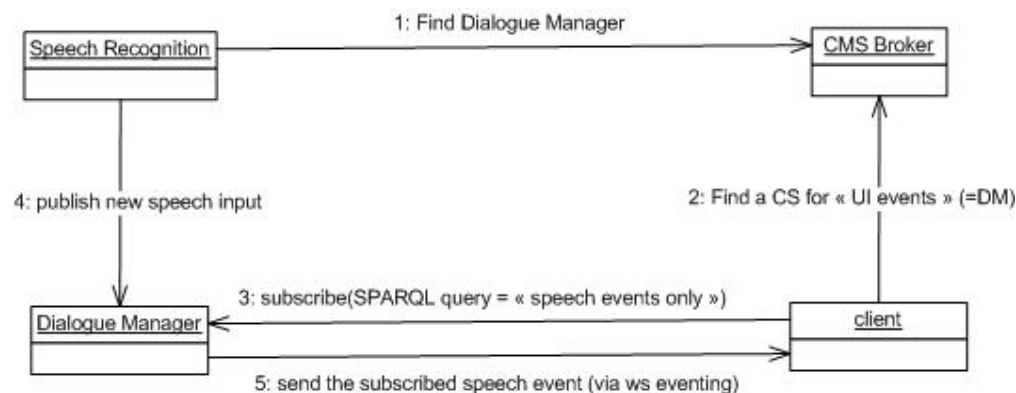


Figure 2. Typical DM user scenario.

Possible issues: The time delay induced by the CMS broker and the web service eventing mechanism may not be acceptable from the user point of view

Fallback solutions: a lightweight subscription mechanism may be developed specifically for the DM

3.3.3.3 Dialogue Manager Internals

The dialogue manager is mainly composed of a blackboard that contains all user inputs and system outputs. Regarding system outputs, a controller checks every current output requests on the blackboard and actually realizes them. This give possibility to solve conflicting outputs on the same terminal, by writing dedicated blackboard knowledge sources that detect and handle such conflicts.

Regarding user inputs, the fundamental UI services like speech recognition simply publish these inputs onto the blackboard, while more advanced UI services shall first subscribe to such events, update/merge them, or even build new interaction events, and publish them back on the blackboard. Advanced UI services can thus handle conflicting user inputs, or solve ambiguous inputs.

Other user inputs clients, typically applications, can thus simply call the subscribe method of the Dialogue Manager and describe the types of user events they are interested in. These clients actually use the same procedure than any classical context consumer, where the details of the user events are encompassed within the SPARQL query.

The main advantage of this blackboard architecture is its flexibility, which makes the development of new advanced UI functionalities easy to realize and to integrate. The blackboard functionality is not directly implemented by the Dialogue Manager Amigo service. The actual blackboard is provided by the PEGASUS framework. The Amigo service provides integration into Amigo and easy-to-use interfaces for other Amigo services.

3.3.3.3.1 PEGASUS

This section provides only a short insight into the architecture of the PEGASUS framework. The complete overview along with a lot of examples and a developer guide for various programming languages is available in the document “Pegasus-Manual.pdf”, which is available at gforge.

The PEGASUS framework introduces an abstract manner of describing a system that consists of:

- Objects of functionality within data contexts
- Communication of objects through trigger concepts and data structures

In the context of this framework, a PEGASUS object implements the following concepts:

- Data Storage
- Triggers
- Functional parts

In PEGASUS, XML is used for the representation of data. A scope of a piece of data reaches from a simple XML element to whole XML trees. In every PEGASUS project, one master application has to hold and maintain the so-called main tree. This tree can be read, manipulated and even extended by other PEGASUS applications. These applications can use subscriptions to be notified upon the change of the arbitrary parts of the XML tree. The main XML tree basically represents the blackboard in PEGASUS. Since various PEGASUS applications may concurrently access parts of the XML tree, the framework has to make sure, that all applications, at all time, are working on the same data. For this purpose, PEGASUS uses distributed XML trees. Access to XML elements is only allowed through PEGASUS's API, which handles the steps needed for concurrent and distributed read and write access on XML elements. This way, application developers can ignore the fact that the XML elements they are

working on are in fact distributed. Figure 3 shows an example of a main tree and 3. Subscribed sub-trees (denoted as *derived trees*).

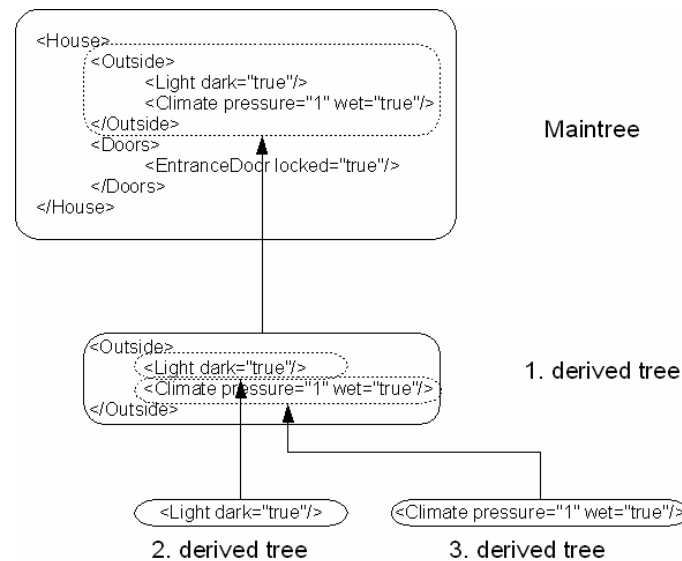


Figure 3. An example XML tree in PEGASUS

To fully implement the blackboard design pattern, it is necessary to enable notification of application on changes in the data storage. PEGASUS provides the concept of *Triggers* for this purpose. A Trigger can be setup to observe a parts of an XML tree and notify application upon a change of that certain element. The changes, which are observed by the Triggers are configurable in PEGASUS. For example, considering the example from Figure 3, it is also possible to define a Trigger that only notifies the change of the element “Light dark” to true but ignores a change to false. Just as well, it is possible to defined Trigger that puts multiple XML elements into a context. For example, notify the changes of “Light dark” only of the element “Climate pressure” holds 1.

For the Triggers to actually cause an action to happen, the functionality of the PEGASUS object needs to be supplied. The functionality of an object is highly dependent on what the respective object represents:

- Source of data. On a system level, it may be a device which expects user input, a logger etc...
- Presentation of data. On an operating system level it may be a public display.
- Process on data. It can either be a concrete program written with the PEGASUS framework using C++ or Java.
- Parts of another process. Object descriptions can be nested, to maintain closeness to abstraction layers
- Abstract representation of one or more processes.
- Way of joining many objects into a container.
- Representation of data structures or input data. It can handle user events, expressed through data change.

For a full description of the PEGASUS API the document “Pegasus-Manual.pdf” is recommended. It contains both, the API for C++ and Java applications.

3.3.3.4 Component interfaces

IContextSource

Containing the following methods:

```
public String query(String contextQueryExpression);
public String subscribe(String contextSubscriptionCharacterisation, String
contextSubscriptionReference);
public boolean unsubscribe(String contextSubscriptionID);
```

IDialogueManager

Containing the following method:

IDialogueManager::publish(Data)

Creates a new entry on the blackboard. This entry contains a user input or a system output.

Note that no knowledge source can actually delete an existing entry on the blackboard. At most, they can mark any entry as “having been processed”, but the actual deletion can only be realized by the blackboard controller.

Every item that is published onto the blackboard shall conform to the following structure, which is undefined yet, but which shall minimally contain the following fields:

- ID: unique identifier (given by the blackboard controller)
- Time: time of publication
- Type: speech input, gesture input, graphical output, ...
- Semantic content: describes the semantic content of the input or output.
- Ambiguous [Boolean]: when the input is ambiguous (e.g. the user command “delete this”), it shall be noted here. The Multi-modal Fusion Module can then try to resolve such ambiguities by combining several.
- ProcessedBy: the list of clients id that have processed this item so far.
- Priority: Several knowledge sources may change the priority of treatment of this item: for instance, when concurrent outputs shall be delivered to the same user.

It may be possible to further provide a “subscription helper” that simplifies the subscription mechanism for external applications: the idea would be to implement a simple subscription() function (with attribute/value matching instead of a full SPARQL query) that activates the complete subscription() method and returns another helper, which can be used by the application to simplify accordingly the SPARQL answer.

3.3.4 Multimodal Fusion

3.3.4.1 Component interface

3.3.4.1.1 Multimodal Fusion Interface

The javadoc for the MFM interface is:

| Method Summary | |
|----------------|---|
| void | <code>cancelUserGesture</code> (java.lang.String timeStamp) Signal that a gesture movement has stopped without any information |
| void | <code>cancelUserSelection</code> (java.lang.String timeStamp) |

| | |
|------|--|
| | Signal that a selection movement has stopped without any information |
| void | <code>cancelUserSentence</code> (java.lang.String timeStamp) Signal that a speech event has stopped without any information |
| void | <code>setAppliConcepts</code> (java.lang.String owlOntology) Set the ontology to be used to test semantic compatibility of objects |
| void | <code>setAppliReferenceableCommands</code> (java.lang.String mmilCommands) Set the commands managed by the application. |
| void | <code>setAppliReferenceableObjects</code> (java.lang.String mmilObjects) Set the objects managed by the application. |
| void | <code>setUserGesture</code> (java.lang.String mmilInputGesture, java.lang.String startStamp, java.lang.String endStamp) Set a gesture's meaning (the mmil component encapsulated in the XML String must have at least one participant) |
| void | <code>setUserSelection</code> (java.lang.String mmilInputSelection, java.lang.String startStamp, java.lang.String endStamp) Set a referential selection event (the mmil component encapsulated in the XML String must have at least one event) - the origin of the selection itself is not relevant but can be expressed (GUI event, Gesture event). |
| void | <code>setUserSentence</code> (java.lang.String mmilInputSentence, java.lang.String startStamp, java.lang.String endStamp) Set a referential expression (the mmil component encapsulated in the XML String must have a participant) |
| void | <code>signalUserGesture</code> (java.lang.String timeStamp) Signal that a gesture movement has started, but is not yet completed |
| void | <code>signalUserSelection</code> (java.lang.String timeStamp) Signal that a selection movement has started, but is not yet completed |
| void | <code>signalUserSentence</code> (java.lang.String timeStamp) Signal that a speech event has started, but is not yet completed |

The format of timestamp parameters is like this: 2006-08-24T14:22:20.358+02:00 (standard XML Gregorian Calendar format, for java, use `javax.xml.datatype.XMLGregorianCalendar`).

The MMIL messages examples are available in the appendix.

3.4 Tutorial

3.4.1 Gesture Service

3.4.1.1 2D Gesture Service

3.4.1.1.1 Using GestureCapture command line

The GestureCapture command line facilitates the scene creation by providing some commands :

| | |
|--------------------|---|
| clear | : clear the scene |
| add <id> | : add a new object from the last drawn simplified gesture, the gesture is simplified (only important points are kept) and added in the scene as an object |
| addfull <id> | : add a new object from the last drawn whole gesture (all the points of the gesture define the object) |
| del <id> | : remove the given object from the scene |
| ls | : list all the objects of the scene |
| col <id> <color> | : color the given object |
| fill/empty <id> | : fill or empty the given object |
| hide/show <id> | : hide or show the given object |
| mul <int> | : multiply int*int times the last drawn object, mainly for debugging purposes |
| back <file> | : load the file as a background image (GIF, JPEG or PNG) |
| save/load <file> | : save or load the scene from the given file |
| javi <file> | : load a JaviMap scene from the given file |
| state <id> | : displays the possible states for an object and its current state |
| state <id> <state> | : change the state of the given object |

<id> is an alphanumeric identifier or could be * for all the objects

<file> is a valid file without spaces, the base directory is the Oscar directory (so the file etc/pic.jpg will be the file \$OSCAR_HOME/etc/pic.jpg)

<color> is either { black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow } or <red><comma><green><comma><blue><comma><alpha> (like "0,123,255,180")

3.4.1.1.2 Creating a scene with JaviMap

It is advised (but not necessary) to use JaviMap to facilitate the scene building, see <http://www4.vc-net.ne.jp/~klivo/mapo/javimap.htm> (Cleve London)

In JaviMap : create an image map from a picture P and save it in JaviMap format.

In GestureCapture :

- load the JaviMap file using command "javi <file>"
- load the background image P with "back <file>"
- save the scene (map+background) using "save <file>"
- edit the scene to add the objType or other features of the objects

3.4.1.1.3 Complete tutorial (with Oscar)

0- achieve the needed requirements, run the OSGI framework (it is normally loaded when you start Oscar) and the required AMIGO services (see the Requirements in the Deployment part)

1- compile each service (gesturecapture, gestureinterpreter, sceneconfigurator) with "ant", and copy the jar files located in the dist directories of each service, to the \$OSCAR_HOME/bundle directory.

2- run the services (**in this order**) :

start file:bundle/gesture-capture.jar

start file:bundle/gesture-interpreter.jar

start file:bundle/scene-configurator.jar

Normally the gesture capture frame and the configuration frame should appear.

3- configure them by first copying the GestureCapture etc directory containing sample scenes into the \$OSCAR_HOME directory. Then push the button "configure" and choose a scene you just copied. You can change the location of the scene files providing that the url they contain remain valid (in the sample scenes they are relative to \$OSCAR_HOME directory). Normally the scene should appear in the gesture capture frame.

4- test some commands. For example type « show * » to show all the objects, type « ls » to list the objects, ...etc.

5- select objects with a mouse or a tactile pen. You can target, point or circle the objects. Check the Oscar windows, you should see that the interpreter sent a GestureSelection event containing all the selected objects associated to salience measures. This GestureSelection event is intended to be consumed by the MultiModalFusion agent, but any module can use it also.

6- now you can use the FillerDemo developed for the AMIGO november 2006 review. This service shows a possible use of the output of the GestureInterpreter : it does nothing else than highlighting for few seconds the selected objects according to the saliences found in the GestureSelection event. Compile the filler-demo service with "ant", copy the jar file in the dist directory to the \$OSCAR_HOME/bundle directory, then run the service :

start [file:bundle/filler-demo.jar](#)

7- then load the apartment scene and selects the objects. You should see that the objects are more or less activated according to your gesture. Note that if you try loading the man

scene, the image will disappear, that is because this scene is built using colored masks and the FillerDemo uses the masks to highlight objects.

8- another feature is the possibility to have sprites representing the objects. An object could have many states associated to different sprites. Load the office scene and type "state laptop" to see the current state and the possible state. Type "state laptop closed" to close the laptop.

3.4.2 Multimodal Fusion component development

The component packages are the following:

- org.amigo.ius.user_interface.multimodal_fusion.impl
- org.amigo.ius.user_interface.multimodal_fusion
- fr.loria.led.mmil.template.jaxb
- fr.loria.led.mmil.template
- fr.loria.led.mmil.objects
- fr.loria.led.mmil.jaxb

Packages under fr.loria.led.mmil are intended to be merged into the main MMIL trunk, but are contained in this repository for now.

Important classes:

- **MultiModalFusionManagerStub** – The stub class for use by clients.
- **MultiModalFusionManagerImpl** – The Component's main class. Basically serves as proxy to the fusionSynchronizer for marshalling/unmarshalling MMILComponents, and filtering the accepted templates.
- **FusionSynchronizer** – Manages a pool of threads waiting for messages occurring in configured time windows, and releasing the messages after time goes out. Calls Fusion.doFusion when messages must be merged.
- **Fusion** – Do the actual fusion of several MMIL components.

3.4.3 Multimodal Fusion application development

In order to give a tutorial for how to use the service from another application, I describe the test client, since it plays the role of several applications sending their output to

After the main bundle of the Multimodal Fusion service is started, one can either use it with the test service or directly. Here I explain how to use the MFM through the description of the test service. The test service exposes a OSGi shell command "scenario" that runs a scenario described in a xml file (either in the test service jar file or via any url) such as:

```
<?xml version="1.0" encoding="UTF-8"?>
<scenario xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://gforge.inria.fr/frs/download.php/894/mmcomm.xsd"
  xmlns:mmil="http://mmil.gforge.inria.fr"
```

```
objects="file:///Volumes/data/Users/guillaume/Documents/workspace/multimodal_fusion_t
ester/res/examples/example-application-objects.xml"
```

```
actions="file:///Volumes/data/Users/guillaume/Documents/workspace/multimodal_fusion_t
ester/res/examples/example-application-actions.xml"
```

```
ontology="file:///Volumes/data/Users/guillaume/Documents/workspace/multimodal_fusion_
```

```

tester/res/examples/home.owl">
  <gestureSignal time="00:00:05"/>
  <gesture start="00:00:05" end="00:00:05.5">
    <mmil:mmilComponent >
      <mmil:event id="e1">
        <mmil:evtType>gesture</mmil:evtType>
      </mmil:event>
      <mmil:event id="e2">
        <mmil:evtType>switchOn</mmil:evtType>
        <mmil:actionStatus>pending</mmil:actionStatus>
      </mmil:event>
      <mmil:relation source="e2" target="e1" type="propContent"/>
    </mmil:mmilComponent>
  </gesture>
  <sentenceSignal time="00:00:04.2"/>
  <sentence start="00:00:05.3" end="00:00:06.4">
    <mmil:mmilComponent xmlns="http://mmil.gforge.inria.fr/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://guillaume.pitel.free.fr/mmil/mmil.xsd">
      <mmil:event id="e1">
        <mmil:dialogueAct>request</mmil:dialogueAct>
        <mmil:evtType>speak</mmil:evtType>
      </mmil:event>
      <mmil:event id="e2">
        <mmil:evtType>switchOn</mmil:evtType>
        <mmil:mode>imperative</mmil:mode>
      </mmil:event>
      <mmil:participant id="p1">
        <mmil:refStatus>pending</mmil:refStatus>
        <mmil:refType>definite</mmil:refType>
        <mmil:objType>CD_Player</mmil:objType>
      </mmil:participant>
      <mmil:relation source="e2" target="e1" type="propContent"/>
      <mmil:relation source="p1" target="e2" type="object"/>
    </mmil:mmilComponent>
  </sentence>
</scenario>

```

Using the “scenario” command is done either with:

- scenario sc01 (where sc01.xml is stored in the jar of the tester, under res/scenarios)
- scenario <url>

The scenario element contains the initialization parameters: ontology, objects and commands files – The ontology is an OWL file, objects and commands are both xml files with MMIL data:

Objects (in res/examples/example-application-objects.xml)

```

<mmilComponent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:amigoApp0="http://amigo.gforge.inria.fr/app0"
  xmlns="http://mmil.gforge.inria.fr/">

  <participant id="room1">
    <objType>objectDesc</objType>
    <modifier>container</modifier>
    <amigoApp0:eltType>Living_Room</amigoApp0:eltType>
    <MMILId>Room3</MMILId>
  </participant>
  <participant id="p1">
    <objType>objectDesc</objType>
    <amigoApp0:eltType>Light</amigoApp0:eltType>
    <MMILId>Light1Room3</MMILId>
  </participant>
  <participant id="p2">
    <objType>objectDesc</objType>
    <amigoApp0:eltType>TV</amigoApp0:eltType>
    <MMILId>TV1Room3</MMILId>
  </participant>

```

```

<participant id="p3">
  <objType>objectDesc</objType>
  <amigoApp0:eltType>VCR</amigoApp0:eltType>
  <MMILId>VCR1Room3</MMILId>
</participant>

<participant id="p4">
  <objType>objectDesc</objType>
  <amigoApp0:eltType>DVD_Player</amigoApp0:eltType>
  <MMILId>DVD1Room3</MMILId>
</participant>

<participant id="g1">
  <objType>groupDesc</objType>
  <amigoApp0:eltType>HiFi</amigoApp0:eltType>
</participant>

<relation source="p1" target="room3" type="contains"/>
<relation source="p2" target="room3" type="contains"/>
<relation source="p3" target="room3" type="contains"/>
<relation source="p4" target="room3" type="contains"/>

<relation source="p2" target="g1" type="inGroup"/>
<relation source="p3" target="g1" type="inGroup"/>
<relation source="p4" target="g1" type="inGroup"/>

</mmilComponent>

```

Commands (in res/examples/example-application-actions.xml)

```

<mmilComponent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:amigoApp0="http://amigo.gforge.inria.fr/app0"
  xmlns="http://mmil.gforge.inria.fr/">

  <participant id="p1">
    <objType>actionDesc</objType>
    <MMILId>switch_on</MMILId>
    <amigoApp0:actType>switchOn</amigoApp0:actType>
  </participant>
  <participant id="p1-1">
    <objType>argumentDesc</objType>
    <amigoApp0:argType>Switch</amigoApp0:argType>
  </participant>
  <relation source="p1-1" target="p1" type="object"/>

  <participant id="p2">
    <objType>actionDesc</objType>
    <MMILId>switch_off</MMILId>
    <amigoApp0:actType>switchOff</amigoApp0:actType>
  </participant>
  <participant id="p2-1">
    <objType>argumentDesc</objType>
    <amigoApp0:argType>Switch</amigoApp0:argType>
  </participant>
  <relation source="p2-1" target="p2" type="object"/>

  <participant id="p3">
    <objType>actionDesc</objType>
    <MMILId>decrease</MMILId>
    <amigoApp0:actType>decrease</amigoApp0:actType>
  </participant>
  <participant id="p3-1">
    <objType>argumentDesc</objType>
    <amigoApp0:argType>Continuous_Scale</amigoApp0:argType>
  </participant>
  <relation source="p3-1" target="p3" type="object"/>

```

```

<participant id="p4">
  <objType>actionDesc</objType>
  <MMILId>increase</MMILId>
  <amigoApp0:actType>increase</amigoApp0:actType>
</participant>
<participant id="p4-1">
  <objType>argumentDesc</objType>
  <amigoApp0:argType>Continuous_Scale</amigoApp0:argType>
</participant>
<relation source="p4-1" target="p4" type="object"/>
</mmilComponent>

```

That's for the initialization part. The actual scenario contains these kinds of elements:

- gesture, gestureSignal, gestureCancel
- sentence, sentenceSignal, sentenceCancel
- selection, selectionSignal, selectionCancel

Each “Signal” and “Cancel” elements are parameterized by a “time” argument stating when to send the corresponding signal to the MFM, based on the start time of the scenario. Other elements have “start” and “end” parameters, for corresponding parameters of the MFM interface’s methods. These methods are invoked at the time specified by the “end” parameter. Those elements contain a MMIL message, which is sent as content in the method calls. As seen in the example scenario, the MMIL component is simply embedded between <sentence> and </sentence> tags (sentence can be replaced by selection or gesture).

When the MFM receives several messages from different modalities (and, actually even if it receives only one message) in a given time window, it tries to unify commands’ references and objects’ references into one unique functional application (with the command as the function and the objects as arguments). Then, it completes the original messages with information from this unique functional application, and publishes all of them at once as an OSGi event. In order to receive this information, services must subscribe to the MFM subscription service for the “fusionEvent” event.

3.4.4 Dialogue Manager

A full usage and developers guide for the backend of the Dialogue Manager (PEGASUS) is available at gforge, in the document called “Pegasus-Manual.pdf”.

3.5 Assessment

3.5.1 Voice Service

3.5.1.1 Implicit Speech Input

The work realized in this subtask mainly concerns two aspects: dialog act recognition and keyword spotting.

Dialog act recognition

This research work has not been integrated into the Amigo middleware and its main outcome is paperwork and stand-alone software. Evaluation is realized in terms of dialog acts recognition rate in French and in Czech and on 2 different tasks: train ticket reservation and broadcast news.

For train ticket reservation, the recognition accuracy is about 93.6 % with 4 dialog acts, and for broadcast news, it is about 77.7 % with 7 dialog acts.

More details can be found in P. Král's thesis ("Automatic recognition of dialogue acts", Pavel Král, Ph.D., Nancy-I University, 2007).

Keyword spotting

This work has been integrated into the topic recognizer, which is evaluated within the WP6 MyNews application. Furthermore, it has been evaluated in terms of keywords recognition accuracy (with speaker dependent models) on a small test set that has been recorded for the multimodal fusion application described next. On this test set, recognition accuracy is 98 % on 50 sentences. The computational complexity of this module is very low, and its speed is about 0.5 times real-time.

3.5.2 Dialogue Manager

A simplified instance of the Dialogue Manager, with limited functionalities, but still following the fundamental design principles exposed above, has been implemented in the framework of the WP6 Role-Playing Game demonstrator, where multiple input and output devices may be used to play a board game.

It proved to be a hard task to implement a generic multidevice-capable Dialogue Manager, which at the same time handles generic applications and is still easy to configure and use. Making the manager less generic, improves the usability, but reduces its capabilities. On the other hand, if the manager is as generic as possible, it is much harder to use for other applications, because the configuration and setup steps for these applications are increased. For these reasons, the only fully implemented application using the Dialogue Manager's blackboard principle is the multidevice-capable board game of WP6.

3.5.3 2D Gesture and Multimodal Fusion

Both modules have been combined into a small demonstration application, where the user may manipulate some items that are displayed in an office picture. In addition, a light speech recognizer based on the keyword spotting module has also been included in order to provide the speech input.

Until now, only informal qualitative assessment has been realized. The conclusions of these experiments are the following:

- Regarding accuracy, both the keyword spotting and 2D gesture capture have a very high recognition rate, at least above 90 %, which can be explained by the very small number of possible keywords and items that strongly constraint the search space. As a consequence, the multimodal fusion module achieves quite correctly its goal and the resulting action matches the expected one at the same level of performance.
- Regarding speed, the overall system is a bit too slow compared to what would have been considered as acceptable: the response delay is about 3 seconds per user action. Neither keyword spotting nor 2D gesture capture introduce significant delays. Most of the delays actually come from the fusion fixed delays, which are basically required when an input in some modality has been detected and that the system is waiting for the next potential modalities. Also, all the modules (keyword spotting, 2D gesture, fusion, application) communicates via the OSGI framework and further access an external service that stores the ontologies. All this is also partly responsible of the computational requirements. But this is just a prototype, and different solutions exist to speed-up the whole procedure, such as externalizing fusion delays to the dialogue manager.

3.6 Appendix

3.6.1 Multimodal Fusion sample files and messages

3.6.1.1 Properties file for time windows:

```
amigo.mmf.timeout.long.selection=2000
amigo.mmf.timeout.long.sentence=2000
amigo.mmf.timeout.long.gesture=2000
amigo.mmf.timeout.short.selection=1000
amigo.mmf.timeout.short.sentence=1000
amigo.mmf.timeout.short.gesture=1000
```

3.6.1.2 MMIL message example for gesture

```
<mmilComponent xmlns="http://www.loria.fr/mmil/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.loria.fr/mmil/
file:/Volumes/data/Users/guillaume/Documents/workspace/multimodal_fusion/res/templates/mmil.xsd">
  <event id="e1">
    <evtType>gesture</evtType>
  </event>
  <event id="e2">
    <evtType>switchOn</evtType>
    <actionStatus>pending</actionStatus>
  </event>
  <relation source="e2" target="e1" type="propContent"/>
</mmilComponent>
```

3.6.1.3 MMIL message example for selection

```
<mmilComponent xmlns="http://www.loria.fr/mmil/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.loria.fr/mmil/
file:/Volumes/data/Users/guillaume/Documents/workspace/multimodal_fusion/res/templates/mmil.xsd">
  <event id="e1">
    <evtType>selection</evtType>
    <actionStatus>pending</actionStatus>
  </event>
  <participant id="p1">
    <set>
      <participant id="p2">
        <MMILId>Light1Room3</MMILId>
        <objType>light</objType>
      </participant>
      <participant id="p3">
        <MMILId>CeilingRoom3</MMILId>
        <objType>ceiling</objType>
      </participant>
    </set>
  </participant>
  <relation source="p1" target="e1" type="propContent"/>
</mmilComponent>
```

3.6.1.4 MMIL message example for sentence

```
<mmilComponent xmlns="http://mmil.gforge.inria.fr/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://guillaume.pitel.free.fr/mmil/mmil.xsd">
  <event id="e1">
```

```

        <dialogueAct>request</dialogueAct>
        <evtType>speak</evtType>
    </event>
    <event id="e2">
        <evtType>ellipsis</evtType>
    </event>
    <participant id="p1">
        <refStatus>pending</refStatus>
        <refType>definite</refType>
        <objType>CD_Player</objType>
    </participant>
    <relation source="e2" target="e1" type="propContent"/>
    <relation source="p1" target="e2" type="object"/>
</mmilComponent>

```

3.7 GUI-Service Component Overview

Nearly every applications needs a way to get in to contact with the user. Typically this is done by a graphical user interface designed for a special application. So the user is presented a graphical interface for his text application, for his painting application, to control the TV or to control the washing machine. Usually, this means that the developer of an application has to write program code for the core service and program code for the GUI. These GUIs are specialised to the problem field their belong to. Unfortunately, the user interfaces of applications that belong to the same field, but produced by different manufactures, will typically differ enormously. As a consequent a user is unfamiliar when changing devices, even if they belong to the same area.

Today and in the future it will be quite usual that users are getting in contact with many different environments equipped with variants services/devices of numerous manufactures. As a result, it will be impossible to bring an Amigo middleware to success if a user has to learn the inner logic of each service each time he is exposed to a new environment.

For this reason, Amigo will use a different way. In Amigo not every service has to implement an own GUI-Application, instead it tries to build one single integrated GUI-Service, which combines every given service to build a homogenous control interface of the environment. The actual GUI is automatically generated by the GUI-Service exploring information provided by the services. Therefore each service has to explain it's capabilities (functionalities) and the internal dependences. The GUI-Service will build an integrated control model out of the numerous descriptions and allows different GUI-Interfaces of different manufactures to enter this model and to make use of it.

This approach makes use of the classic MVC (model view control) approach. The descriptions and the related services represent the model. The control is done by the GUI-Service, while the views are represented by the realised GUI-Interfaces connected to the GUI-Service.

Provider

IMS

Introduction

Development status

Intended audience

Project partners

Developers**License**

LGPL

Language

Java

Environment (set-up) info needed if you want to run this sw (service)

Windows XP, JDK1.5, OSCAR, Pellet 1.4

Platform

JVM, OSGi-Based Programming & Deployment Framework from WP3

Tools

Oscar

Files

Available for download at

[amigo_gforge]/ius/user_interface/gui_service/

Documents**Tasks****Bugs****Patches**

3.8 Deployment GUI Service

3.8.1 System requirements

OSGI framework

OSCAR platform (can be on either linux or windows)

Refer to [OSCAR req]

3.8.2 Download

Source code available for download at

[amigo_gforge]/ius/user_interface/gui_service/

3.8.3 Install

First you have to start pellet on port 12335.

The GUI-Service can be installed as any other bundle into OSCAR. Then you have to install and start the following bundles in the following order:

1. gui.libs
2. gui.Knowledgebase
3. gui.core

If you want to start the example graphical user interface you have to start the following bundles:

1. gui.ehome_app_driver_api
2. gui.ehome_app_driver
3. gui.application

If you want to log in users for testing the gui application, you can install:

- gui.GUIControlClientGUI

Faked devices for testing can be added by installing and running the bundle:

- gui.ServiceFaker

3.8.4 Configure

In the bundle three test users are configured, which could be directly used (other can be added or removed). The personal navigation strategies of these users are defined by the files Strategy[1-3].xml. The files can be found in gui.core/ExampleMenuStrategies.

3.9 Component Architecture GUI Service

The architecture of the GUI service is shown in Figure 4.

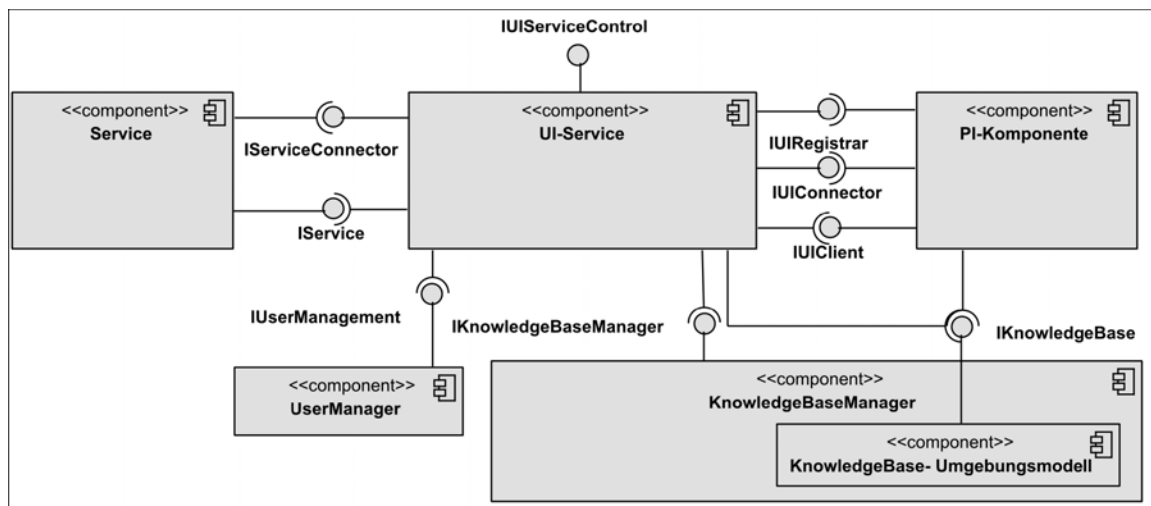


Figure 4 Components of the complete GUI service

The core components of the GUI service are

- **Services and devices:** The actual services and devices embedded in the environment implement the various functions. They combine sensor input with user requests. The services and devices form the actual application intelligence within the environment. They register with the UI service in order to be able to interact with the user. Hence, they do not directly interact with the user but just describe their requirements and possible options the user may choose. To this end, during registration, ontology based descriptions of the services are transferred to the UI service.
- **A user interface service:** The user interface service (UI-service) provides the actual control mechanisms for the presentation and interaction components of the system. Further, it controls the interaction of user and system and also handles the different user profiles. The UI-service determines the actual content and navigation options of the user interface. Both elements are generated based on the navigation model associated with the corresponding user. Content and navigation model are transmitted to the presentation and interaction components. Further, the UI-service receives and forwards events that are received from services. The input that is generated by the user is processed by the UI-service and is either forwarded to the specific services or devices or is used to control navigation within the user interface.
- **A knowledge base manager:** The knowledge base manager holds the entire knowledge about the environment. This includes information about available services and devices. As a result, each device transmits its knowledge base that covers the device's functionality to the manager. The manager takes the various knowledge bases and builds a large information database from it. The UI-service uses this integrated knowledge based in order to determine which information and what navigation structures shall be displayed.
- **A user manager:** The user manager takes care of the various navigation strategies of the different users. Note that this information is stored and retrieved using UMPS.
- **Presentation and interaction components (PI-component):** These components are responsible for displaying the actual information and receiving the user input. They register with the UI-service and receive the information that are to be shown. For Amigo, a Java based PI-component has been implemented. In order to display a

specific piece of information, a presentation repository is used. This repository stores for a defined set of information types how they shall be shown on the screen. The association of information type and the actual presentation on the screen is done via a presentation map. Searching the map is based on the environmental model.

3.9.1 Interfaces

Figure 4 and Figure 5 show the interfaces that are used within the UI-service.

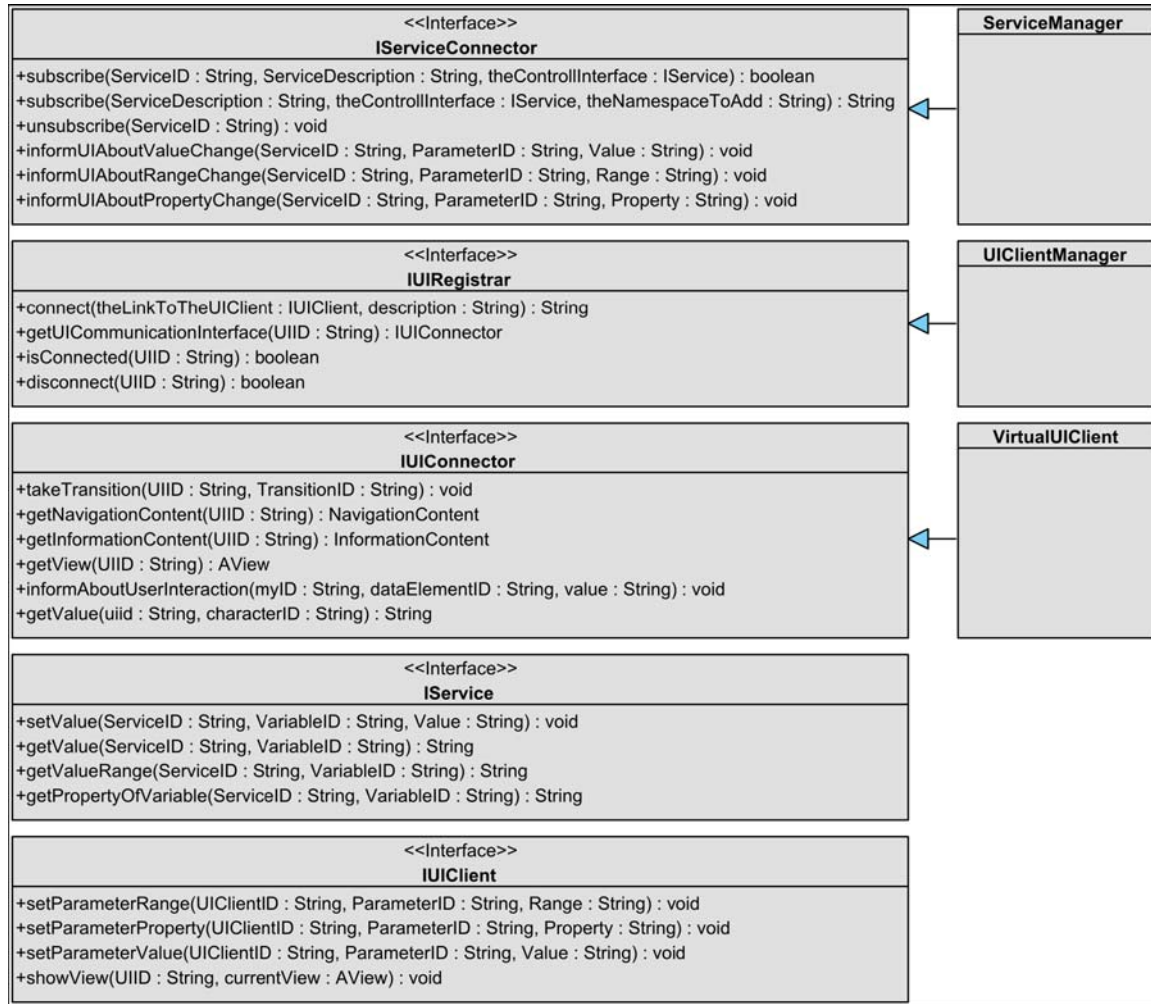


Figure 5 Interface definitions of components within the UI-service

The interface between the UI-service and devices or services is **IServiceConnector**. By using this interface, services can register with the UI-services. This interface is implemented by class **ServiceManager**. It is also used by services and devices to inform the system about changes. Further, services and devices may use this interface to deregister themselves from the system. In order to receive input generated by the user, the **IService** interface must be implemented. Moreover, this interface can be used to retrieve values and valid value ranges.

The interface **UIRegistrar** is implemented by class **UIClientManager**. PI-components use this interface to register or deregister themselves from the UI-service. For each PIO-component a separate instance for the class **VirtualUIClient** is created during registration. The instances provide interfaces of type **IUIClientConnector**, which is responsible to establish communication between PI-components and the UI-service. PI-components use this interface to transmit user

interface inputs. They can also retrieve information about a current view that is active in the user interface. Further, PI-components must implement interface `IUIClient` in order to receive information about state changes, changes with respect to the valid value range or to be informed when a specific piece of information is not shown. Moreover, the IP-components are informed via this interface what information is to be shown on the screen and which navigation options the user has.

Interface `IUserManagement` is used to attach user management components. From the user profile the user specific navigation model is retrieved.

Interface `IKnowledgeBase` is exploited to interface knowledge bases to the system. Via this interface, new knowledge can be added to the system. This covers adding concepts and statements. Finally, information can be retrieved from the knowledge base via this interface as well.

The interface `IUIServiceControl` can be used to control the behavior of the UI-service. For example, using this interface, PI-components can be coupled, user can be logged in or out and statements can be added to the environmental model. If PI-components are coupled, they behave the same. I.e., they show the same information to the user and also present him or her with the same navigation options. This may be useful if there are several displays in the range of the user giving him or her the options to freely choose between them. Moreover, the user may even change the interaction device on the fly.

3.9.2 Classes of the UI-Service

Figure 6 shows the main classes of the UI-service. Class `UIMain` implements the singleton pattern. The appropriate instance is created via the `Activator` class. `UIMain` creates further objects needed by the service. Important classes of the UI-service are:

- *Service Manager*: The service manager also belongs to the singleton pattern. It is responsible for administration and communication with services and devices. Services inform the `ServiceManager` about state changes using class `ServiceActionManager`.
- *ServiceActionManager*: There is also only a single instance of class `ServiceActionManager`. The class receives information about changes of internal variables of services or devices. The instance then determines those PI-components that are affected by the propagated change and further propagates the information to other components.
- *InternalServiceList*: This class is used to manage the registered services.
- *UIClientManger*: This class implements the interface to register PI-components. For each registered component this class generates a `VirtualUIClient` instance which is used to control the component.
- *VirtualUIClient*: Instances of the class `VirtualUIClient` represent PI-components. Hence, for each PI-component an instance is created to control the component. Class `VirtualUIClient` implements `IUIConnector` which is used for communication with the associated PI-component.
- *UserInputManager*: All user input is processed by an instance of `UserInputManager`. If the user is operating a navigation element, then the instance informs the corresponding instances of class `VirtualUIClient`. Otherwise, if the input changes a value, then the input is forwarded to the appropriate instance of class `ServiceManager`.
- *UIServiceController*: This class can be used to control the behavior of the UI-service. For example, PI-components can be coupled, user can be logged in or out and statements can be added to the environmental model.

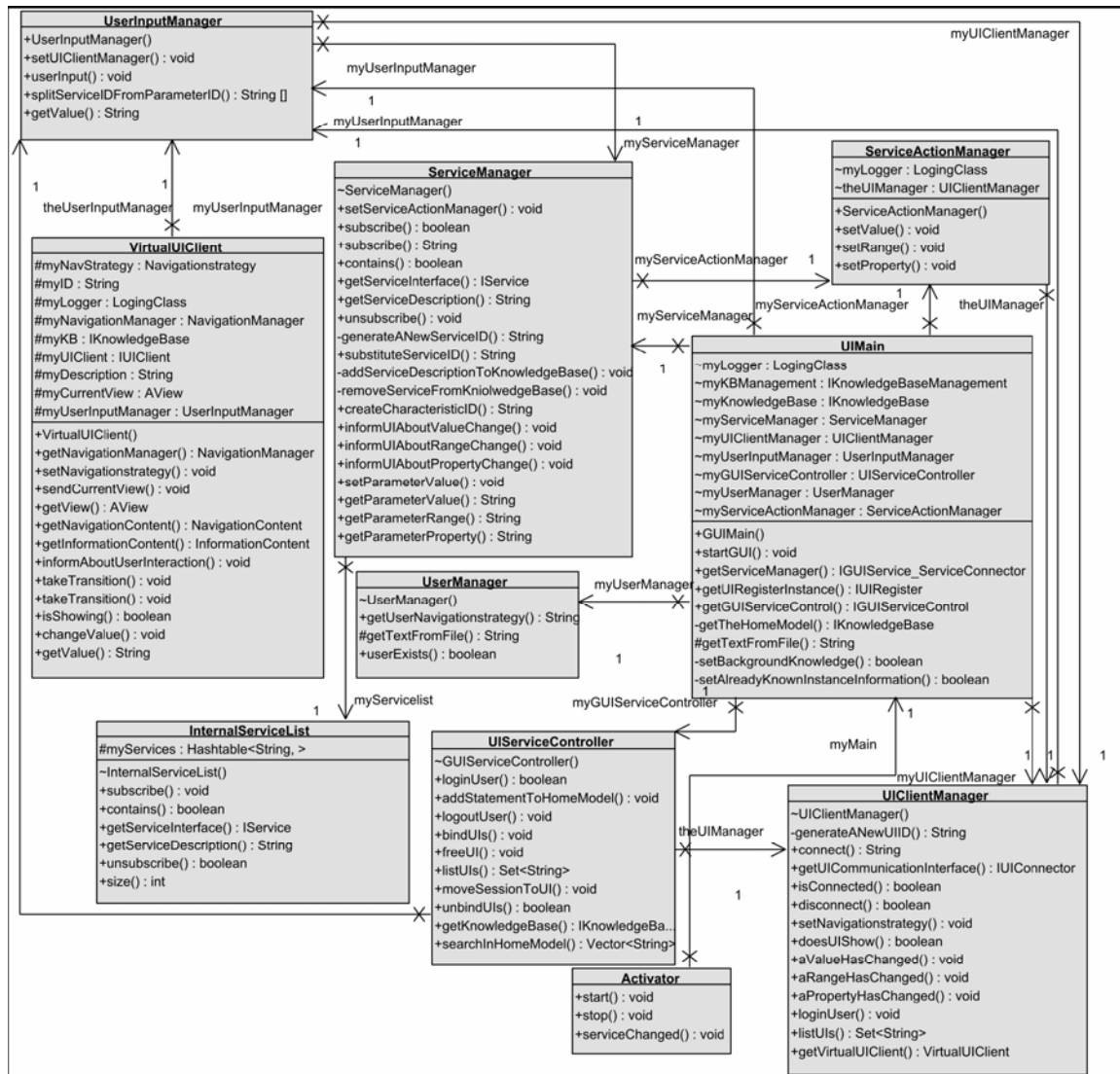


Figure 6 Core classes of the UI-service

3.9.3 Classes used for navigation

Each instance of the class VirtualUIClient is also associated with an instance of class NavigationManager. The Manager is responsible for the navigation options and also determines what information are provided to the user during navigation. To this end, it includes an instance of class NavigationStrategy. This class holds the abstract views, information units and the navigation units. The description of the personal navigation model is stored in XML format. According to this description, the instances of the various classes are derived. The relation of these classes are shown in Figure 7.

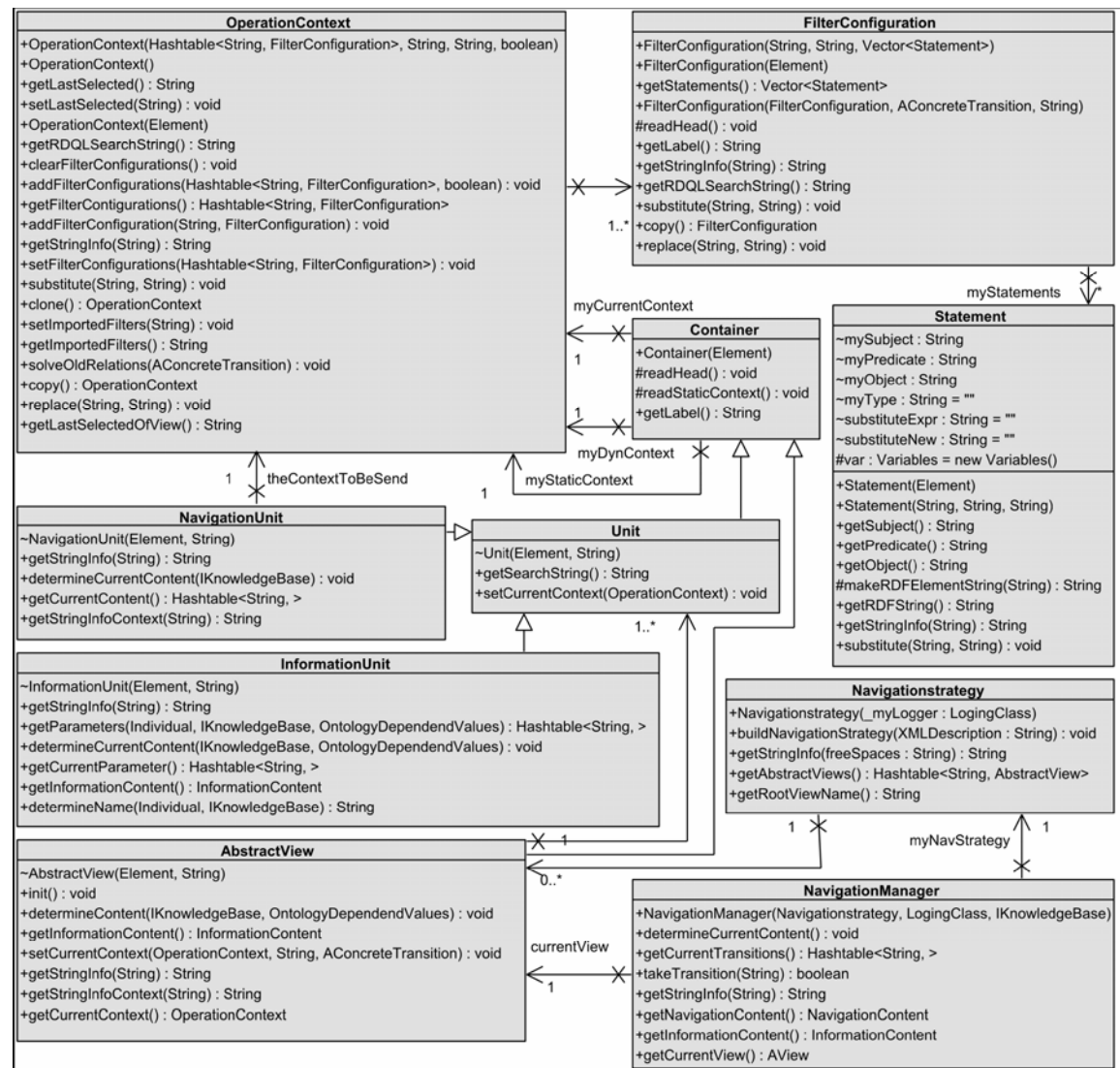


Figure 7 Classes used for navigation

3.10 Assessment GUI Service

3.10.1 User Acceptance

As part of the work package 4.7.2 a GUI mock-up has been developed. The goal of this mock-up was to find out how the menu structure of a graphical user interface shall look like and how people interact with GUIs that follow different navigation strategies.

To this end, we built three mock-ups that all represent a different way how a menu structure is logically generated. Each mock-up represents a menu structure for the same simple home that consists of several rooms (bath room, kitchen, hall, living room, bedroom) with various devices (e.g., cooker, TV set, radio, windows, light). The difference between the mock-ups were that the first two mock-ups share a device oriented view while the last mock-up is function oriented. That is, in the last mock-up the device borders are dissolved bringing functions of different devices onto a single menu page.

In order to find out how user operate with these different kind of menu systems, the mock-up were implemented as Java applications that can be executed on a PC. The test user were assigned a set of tasks they had to execute. The time required by the various users were recorded. Further, the user had to fill out a questionnaire.

The current section is a summary of the deliverable for work package 4.7.2. Please refer to this document for further details about the user acceptance study.

3.10.1.1 Study Methodology and Hypothesis

Our tests aimed at analyzing the effectiveness of different menu structure from a use perspective. That is, the outcome of the tests is expected to answer crucial questions regarding user preferences on menu structures. To obtain the results, users were confronted with the mock-ups and asked to perform a set of task. The results of this user test will feed into development of the user interface services.

One major goal of the test was to find out, how people would like to have their menu structures logically laid out. Especially, we want to check out, whether people prefer a device/room oriented view on the devices or prefer to focus on the functions instead of devices.

Before the tests, we had the following expectations:

- It is expected that users prefer different user interfaces and that the preference is somehow equally distributed among the menu candidates. I.e., we expect that there is no single “one approach fits all” solution
- We expect that user become faster when they become used to the specific way a menu structure is logically build.
- We expect that user will prefer those menu structures which they are able to reach their goal in less time.

3.10.1.2 Test apparatus

There were three mock-ups created for the same virtual apartment. The mock-ups were implemented as Java based applications. Hence, user were able to operate it via a mouse.

3.10.1.2.1 Mock-up 1

The first mock-up implemented a menu structure where at the first menu level the user had to choose the appropriate room, then a device and finally a function of the device. Depending on the device type, they could then directly select a specific function or had to choose between

different sub-menus like “basic function” or “recording”. This menu structure is classified as “device” centric in the following.

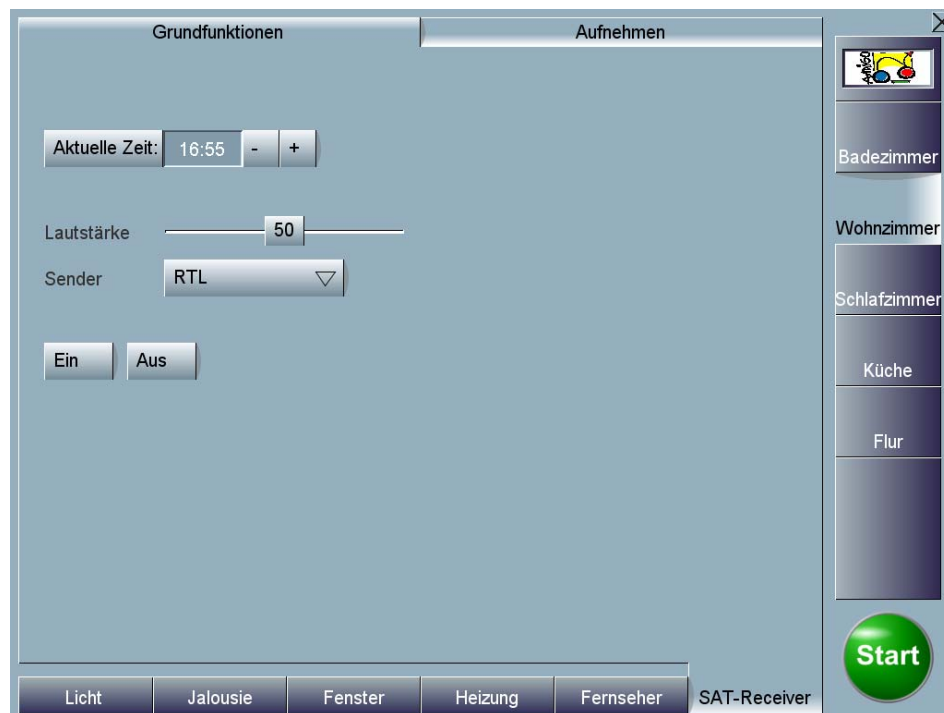


Figure 1: Menu structure of the first mock-up

Figure 1 shows a screen shot of the first mock-up. As can be seen, the first menu level is displayed on the right hand side (room selection). Then, the devices in the appropriate room become available (bottom of window). Finally, the user can select the appropriate function or chose between “basic functions” and “recording functions” (top of window).

3.10.1.2.2 Mock-up 2

The second mock-up implemented a menu structure that allows the user to choose between ambient functions, comfort function, multimedia and white goods. In the next level, the user selects between different rooms. In case of a complex device, the user selects between “basic function” and “recording”. Finally, the actual device function can be chosen.

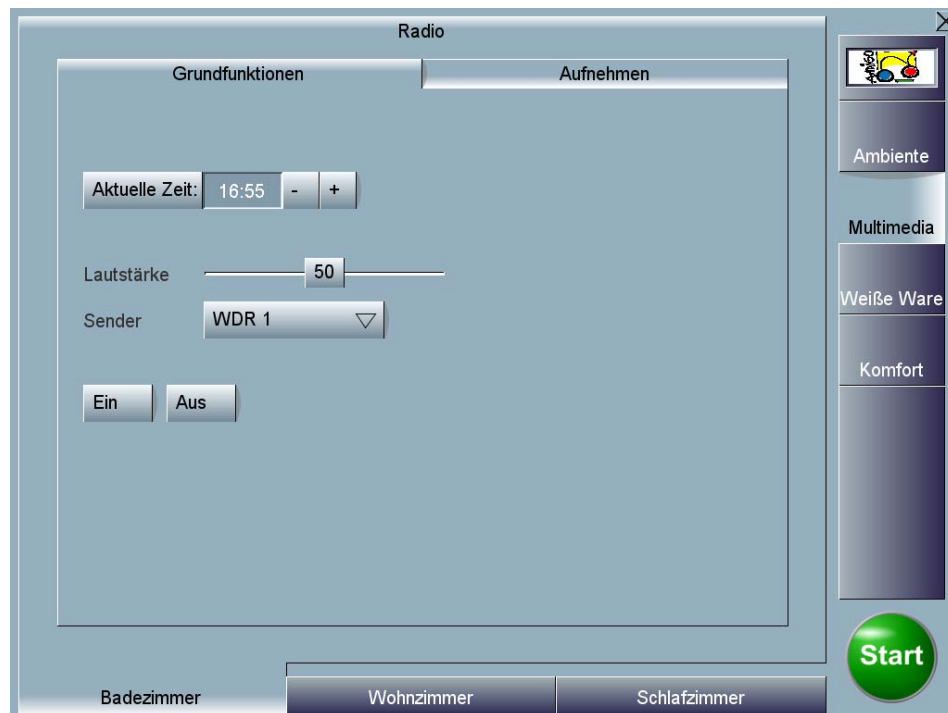


Figure 2: Menu structure of the second mock-up

An sample screen shot of the mock-up is shown in Figure 2. Again, the first (top) menu level can be selected at the right hand side (category of the device). Then, available rooms are selected on the bottom page. Due to the previous selections, the devices available in a room and belonging to the selected category are shown and can be selected via the menu items shown on the top. Finally, additional sub-menus may be selected depending on the complexity of the devices. This menu structure also belongs to the “device-centric” class.

3.10.1.2.3 Mock-up 3

At the first menu level, the functions are grouped by ambient functions, comfort function, entertainment, wellness etc. Then, rooms can be selected. Afterwards, for complex devices the functions are separated by functions that are related to setup, programming and “life”. “life” functions have a direct effect on the behavior of the home.



Figure 3: Menu structure of the third mock-up

The third mock-up is shown in Figure 3. Note that in contrast to the other both mock-ups a function centric view is presented. That is, the user first selects a function category (e.g., entertainment or cooking) and then chooses the appropriate room. Next, in top part of the menu screen the user can do further selection on the functions.

An sample screen to show that on the final page there may show up functions of more than a single device is Figure 4. Here the “live” functions for the TV-set as well as for the satellite receiver are presented on a single page.

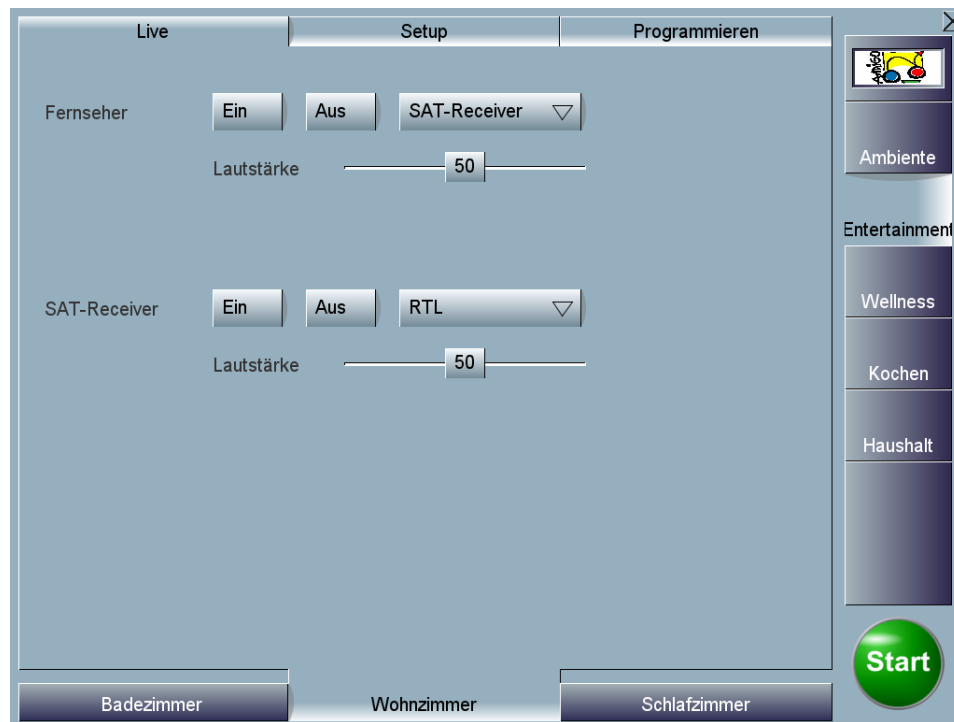


Figure 4: Final menu page with functions of two devices (TV-set and satellite receiver)

3.10.2 Test Results

An important question is what people think about menu structures of current devices and how these structures typically relate to each other. Hence, people were asked whether they think that the handling of current devices differ significantly or not. In Figure 5 the results show that the vast majority of people think that varying menu structures are actually an important issue.

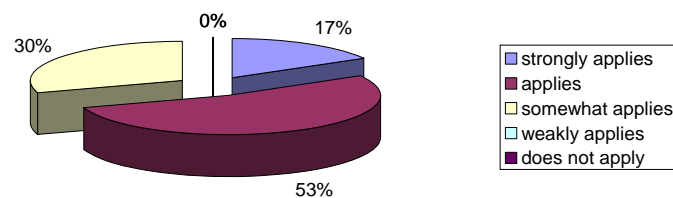


Figure 5: Handling of modern devices differs

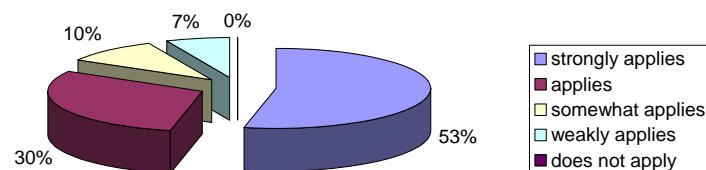


Figure 6: Need for a single user interface

Further, people were asked about the advantage of having a single user interface to their devices (see Figure 6). Obviously, the need for a single homogenous user interface for all (or many) devices in the home is very high. This is not surprising, as struggling with many remote controls is quite a common scenario in today homes.

One of the most important question is of course about the preferred menu structure. As can be seen from Figure 7, the most popular menu structure is the simple “room-> device->function” (device centric) approach. That is, 2/3 of the people prefer to look at the devices from a hierarchical point of view. However, note that still 34% prefer another logical menu layout. The strong bias towards this simple menu structure is somehow surprising. A reason for this might be that people are currently used to look at functions from a device perspective because this is the currently the way functions are organized. As a result, this preference may change if more and more functions are embedded into the environment where no single device can be identified to be the host of a specific function.

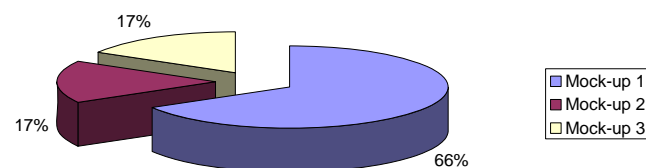


Figure 7: Preferred menu structure

In order to have more insight into the reasons why people prefer the menu structure 1 see Figure 8. Obviously, a big advantage of the simple menu structure is that people can more easily grasp the core idea that has been used to build it.

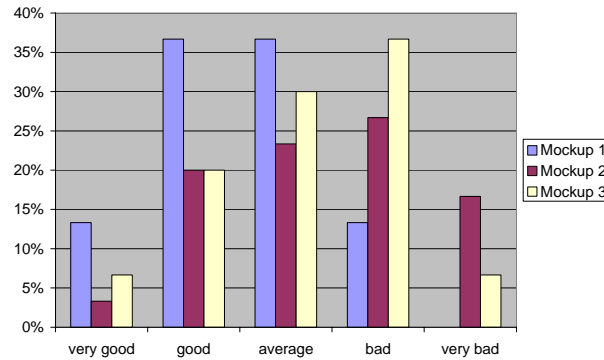


Figure 8: Can the menus be operated in an intuitive way?

Note that efficiency does not play the first role when deciding about the preferred menu system. In the following figure, the efficiency ratings are presented. While the first menu structure is still the winner, the difference especially between the first and the second structure is not as big as the overall preference might indicate.

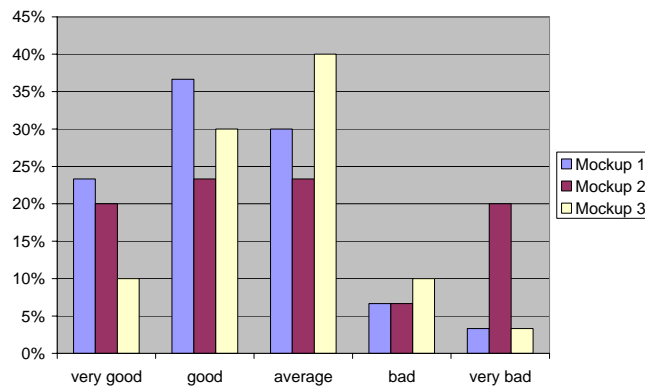


Figure 9: Efficiency of the menu structures

In the figure below, the rating for the conciseness of the mock-ups is shown. Here, menu structure 1 can claim a significant better acceptance than the other approaches. Hence, conciseness seems to be a more important aspect of a GUI than the time needed to fulfil a specific task.

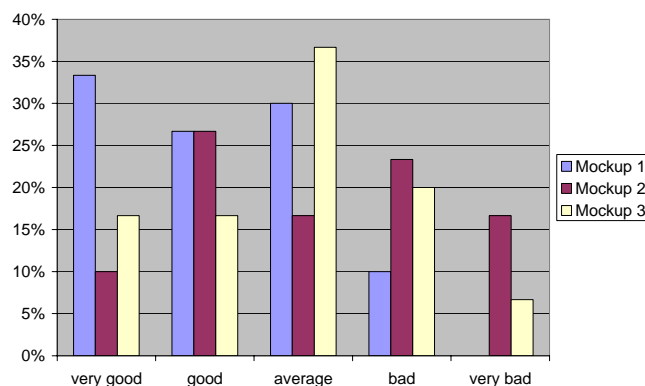


Figure 10: Conciseness of the menu structures

To backup this qualitative results we also done some quantitative experiments where the execution time for the various experiments were measured and compared. In general, the results show that people are getting faster when they become used to a user interface. However, the results also showed that usage speed or efficiency is not the major criteria to rate a specific menu structure.

3.10.3 System Performance

Our experiences show that adding devices is a time consuming task as the knowledge base has to be updated in order to take the environmental changes into account. Thereafter, the menu structures for the various user are synthesized off-line.

The advantage of this approach is that the actual interaction of user and system is not subject to any significant delays as all menu structures and navigation options have been generated already off-line. However, the system may suffer from delay that is caused by the actual services and devices that are controlled by the UI. I.e., the overall performance of the system is significantly affected by the performance of the devices and services that are operated by the user via the UI. Hence, a general statement about the overall performance of the GUI cannot be made without taking the actual devices and services into account.

Nevertheless, when neglecting any delay caused by the controlled devices and services, it can be stated that the GUI shows a sufficient performance when executed on a typical office PC. A more detailed analysis is now ongoing. However, results cannot be published yet.

4 Appendix