ALIZ-E

Adaptive Strategies for Sustainable Long-Term Social Interaction
EU FP7 project, Grant Agreement no. 248116 Seventh Framework Programme (FP7)

Objective ICT 2009.2.1: Cognitive Systems and Robotics

Deliverable D6.6
Documented open-source API

Deliverable submission: Month 54
August 2014

Contents

1 ALIZ-E Integrated System Overview **4**
1.1 Purpose of the System . . . 4
1.2 Interactive Activities . . . 4
1.3 Extra-Activity Robot Behaviour . . . 5
1.4 Robotic Theatre . . . 5
1.5 Wizard Operation . . . 5

2 ALIZ-E Component Architecture **7**
2.1 Functional Architecture . . . 7
2.2 Component Hosting . . . 7

3 Urbi Middleware **9**
3.1 Urbiscript Scripting Language . . . 9
3.2 UObject Components . . . 10
3.2.1 UObject C++ . . . 10
3.2.2 UObject Java . . . 10
3.3 Urbi App for NAO . . . 10

4 Installation **11**
4.1 Material Required . . . 11
4.2 Initial Robot Setup . . . 11
4.3 Initial Computer Setup . . . 11
4.3.1 Requirements . . . 11
4.3.2 ALIZE Checkout . . . 12
4.3.3 Urbi 3 . . . 12
4.3.4 MARY TTS . . . 12
4.3.5 Julius ASR . . . 13
4.4 Compile, Install and Deploy ALIZ-E . . . 13
4.4.1 Compilation . . . 13
4.4.2 Deploy on the Robot . . . 13
4.4.3 Install the Dance Resource Server . . . 14
4.5 Run ALIZ-E . . . 14

5 Components **16**
5.1 Wizard GUI . . . 16
5.2 Activity Manager . . . 17
5.3 Level 0 . . . 17
5.4 Basic Awareness . . . 17
5.5 Dialogue Manager . . . 17
5.6 Motion Generation . . . 17
5.6.1 NaoConfig . . . 18
5.6.2 Scripted Movements . . . 18
5.6.3 Perlin Noise . . . 18
5.6.4 Body Openness . . . 18
5.6.5 Dance Movements Generation . . . 18
5.7 Automatic Speech Recognition . . . 18
5.8 Face Emotion Recognition . . . 18
5.9 Kinect Recognition . . . 19
5.10 Memory System . . . 19
5.11 Motion & Speech Synchronizer . . . 19
5.12 Text to Speech (TTS) . . . 19

5.13 User Model 20
5.14 Voice Activity Detection (VAD) 20
5.15 Voice Modification 20

6 Cloud computing platform **22**
6.1 Cloud versus Embedded Services 22
6.2 Guidelines 22
6.3 Embedded Fall-back 23

7 Annexes **24**
7.1 Activity Manager README 24
7.2 Creative Dance 27
7.3 Creative Dance README 31
7.4 Creative Dance Server README 32
7.5 Face Emotion Recognition 34
7.6 Kinect recognition README 39
7.7 Memory API 41
7.8 Speech Recognition README 52
7.9 Text-to-Speech README 56
7.10 User Model README 58
7.11 Voice Activity Detection README 62
7.12 Voice Modification README 67

1 ALIZ-E Integrated System Overview

1.1 Purpose of the System

The ALIZ-E Integrated System is a software system that was developed for the NAO Robot in the context of the ALIZ-E European project (`http://aliz-e.org`). Our goal was to create a new generation of robotic systems able to establish socio-emotional relationships with their users, through a range of applications, including edutainment and medical care for young patients.

The software system was used in the context of several experiments to study the interaction between diabetic children and a NAO robot. During the experiments children could interact with the NAO robot and play three different interactive activities with it. The activities were a quiz game using a touchscreen tablet ('Quiz Game'), a food sorting game on a touchscreen table ('Sandtray'), and a creative dance game ('Creative Dance').

This deliverable will present the architecture of the system we designed, its applications, and the necessary documentation to install and reuse the different components of the system.

1.2 Interactive Activities

In order to enable interaction between a child and a NAO robot the ALIZ-E system integrates three interactive activities. Each interactive activity is a high-level piece of software that allows the NAO robot to interact with a child towards a specific goal. The three interactive activities that are implemented in the ALIZ-E Integrated System are as follows:

1. **Quiz Game** - In the Quiz Game, a NAO and a child take turns in asking each other questions presented on a see-saw tablet (Figure 1). The game comes with a database of questions in Italian which focus on general knowledge of diabetes, daily routines and self-support.

2. **Sandtray** - The Sandtray is a large touchscreen table which can display a variety of sorting games, for example, sorting foods into 'high carbohydrate' or 'low carbohydrate' groups (Figure 2). Two interlocutors (in this case a child and a robot) can play simultaneously without a specified interaction structure.

3. **Creative Dance** - In Creative Dance the NAO teaches a physical activity routine to a child (Figure 3).

Figure 1: Child playing the Quiz Game with a NAO robot.

Figure 2: Child playing with the Sandtray.

Figure 3: Child playing the Creative Dance.

1.3 Extra-Activity Robot Behaviour

The robot should have a coherent overall behaviour, and appear to be 'alive' all of the time. To this end, when it is not interacting in an activity with a child, it should not be static or non-responsive.

The ALIZ-E system incorporates components that will keep the robot active between activities and sessions. We provide a minimal behaviour, called 'Level 0'. In this mode the robot remains seated, and looks around when it hears sounds. If children are present then the robot will look at them and follow their motion with its gaze. The goal here is to provide the robot with a sense of agency and to pro-actively seek interaction.

1.4 Robotic Theatre

The ALIZ-E system is used in a 'Robotic Theatre' (Figure 4). This is a space in which all equipment required for the different activities is laid out. This includes a tablet for the Quiz Game, the Sandtray and an additional screen for Creative Dance.

1.5 Wizard Operation

The ALIZ-E system provides a complete Wizard interface to control the behaviour of the robot during experiments. It supports the system by allowing experimenters to provide input when the

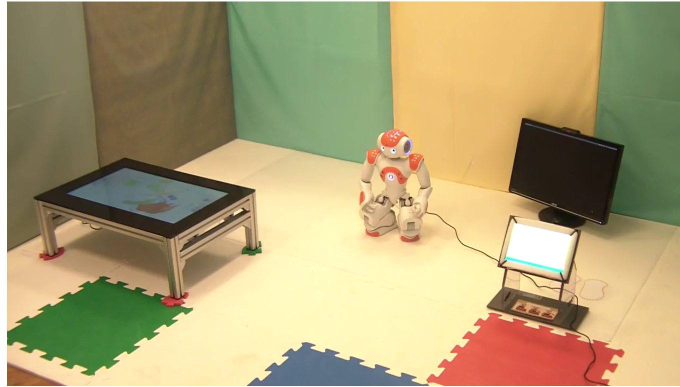

Figure 4: Robotic Theatre as used in the ALIZ-E Integrated System.

system is not able to make an autonomous decision. With the Wizard interface, an experimenter can decide to start or stop an interactive activity, and also control the internal parameters of activities. The Wizard interface can be used to change the position of the robot, and allow it to navigate the play room in order to attend to different game elements (the tablet, the touchscreen table).

2 ALIZ-E Component Architecture

The system is composed of multiple components that assume a specific role in the overall system. The low-level components used within the system are utilised in such a way that the NAO can exhibit a global, coherent behaviour across all interactive activities. Here we provide a general overview of the system, and how the components fit within it.

2.1 Functional Architecture

From a functional perspective, several coarse functional layers can be identified within the system (Figure 5):

1. **Service Layer** - The components in the Service Layer add new capabilities to the robot, such as speaking, understanding speech, remembering information about a user, and so on.

2. **Activity Layer** - In this layer the components implement a complete behaviour for each of the interactive activities, plus the Level 0 behaviour for time between activities.

3. **Control Layer** - This layer contains components dedicated to the control of the overall behaviour of the robot. It is responsible for orchestration between the different activities present in the Activity layer, starting and stopping them at the appropriate times. The experimenter Wizard GUI is also part of this layer.

Figure 5: Components of the system organized by functional layers.

The capabilities of the system can be easily expanded by adding more service components into the Service Layer. It is also easy to add new interactive behaviours for the robot by adding more behaviour components in the Activity Layer.

2.2 Component Hosting

Not all components of the system run on the robot. The majority of components run on a separate computer and connect to the Urbi middleware running on the robot. Not all components run under the same Operating System; Figure 6 shows how these components are distributed across multiple

computers. Some components run under Ubuntu Linux, while the emotion recognition component requires a computer with Windows 7.

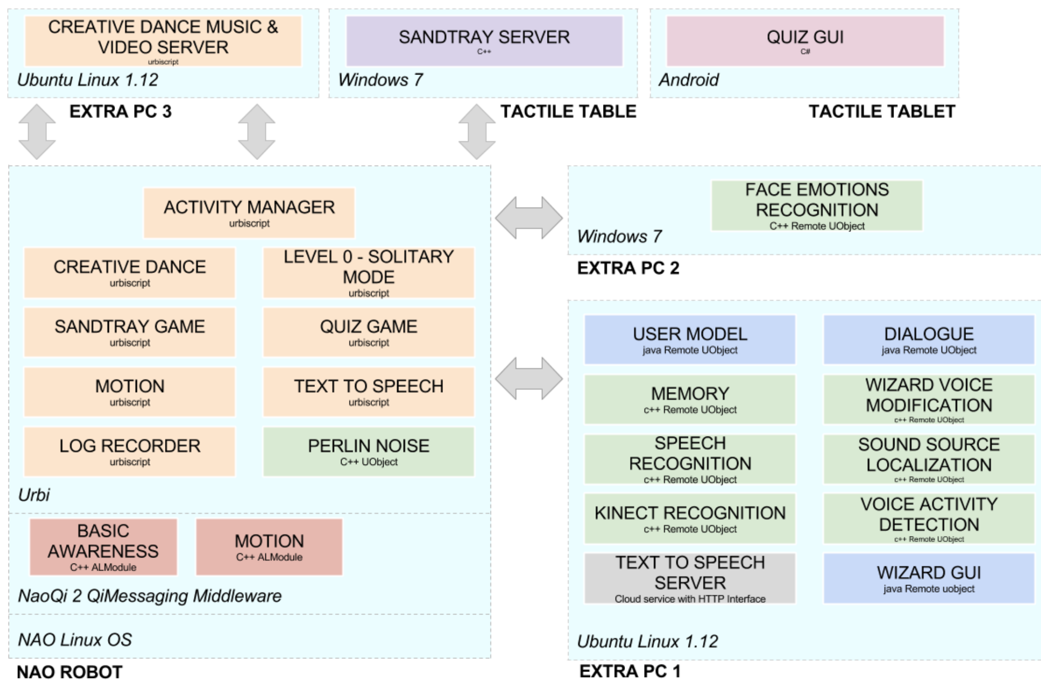

Figure 6: Component hosting and technologies used for each.

The different components were developed using several programming languages: C++, Java and urbiscript. The Urbi middleware acts as an orchestrator at the core of the system, enabling communication between all of the components.

3 Urbi Middleware

We relied on the Urbi middleware to support the integration of all software components on the NAO robot. Urbi is an open-source software platform for robotic and complex systems, developed by Gostai/Aldebaran. The Urbi SDK is a fully-featured environment to orchestrate complex organizations of components. It relies on a middleware architecture that coordinates components named 'UObjects'. It also features urbiscript; a scripting language that can be used to write orchestration programs.

Urbi was first designed for robotics; it provides all of the necessary features to coordinate the execution of various robot components (actuators, sensors, software devices that provide features such as text-to-speech, face recognition, and so on). On the NAO robot, Urbi runs on top of the NaoQi Operating System (based on Linux). It interfaces with NaoQi libraries in order to communicate with NAO hardware, and to benefit from all of the Naoqi behaviours (Figure 7). The Urbi middleware sources are freely available on GitHub, at `https://github.com/aldebaran/urbi`.

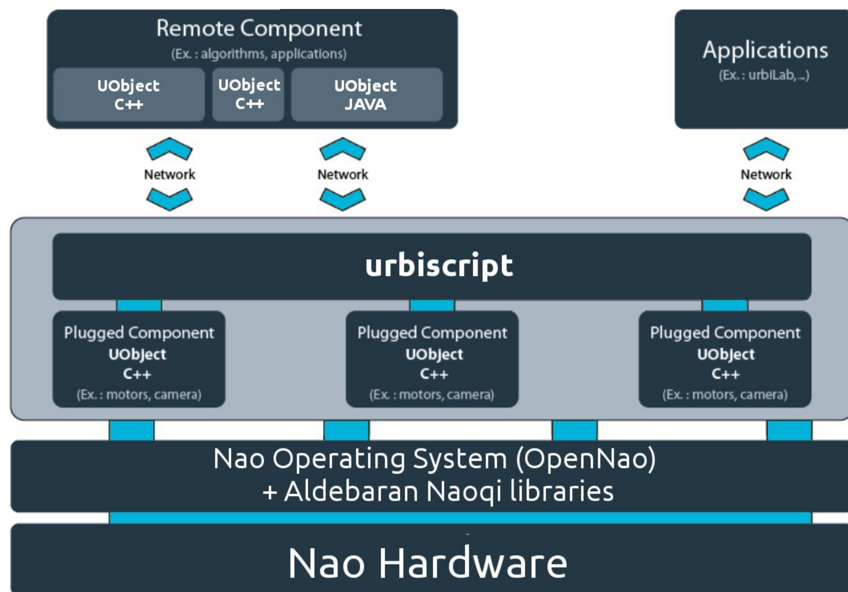

Figure 7: Urbi architecture.

3.1 Urbiscript Scripting Language

Urbiscript is a programming language primarily designed for robotics. It's a dynamic, prototype-based, object-oriented scripting language. It supports and emphasizes parallel and event-based programming, which are very popular paradigms in robotics, by providing core primitives and language constructs.

Its main features are:

- syntactically close to C++.
- integrated with C++, Java, Python.
- object-oriented.
- concurrent.
- event-based.
- functional programming.
- client/server.
- distributed.

3.2 UObject Components

Urbi makes the orchestration of independent, concurrent, components easier. It makes it possible to use remote components as if they were local, thereby allowing concurrent execution, and synchronous or asynchronous requests. Such components are called 'UObjects'. Components need not be designed with UObjects in mind, rather, UObjects are typically "shells" around "regular" components.

3.2.1 UObject C++

The UObject C++ architecture provides the API to write C++ components and use them in highly concurrent settings. As soon as they provide a UObject interface, one can interact with these components (making queries, changing them, observing their state, monitoring various kinds of events and so on) through Urbi. To summarize, the UObject API can be used to add new objects written in C ++ to the urbiscript language, and to interact from C++ with the objects that are already defined. Controlling a physical device (servomotor, speaker, camera), and interfacing higher-lever components (voice recognition, object detection) are performed by Urbi.

3.2.2 UObject Java

The UObject Java API is the matching piece of the UObject C++ API. It is generated from the C++ API, and relies on a native C++ library. It can be used to add new remote components written in Java to the Urbiscript language, and to interact from Java with the objects that are already defined. Only the use cases of interfacing higher-lever components (voice recognition, object detection) with Urbi are performed using the Java language.

Compared to the UObject C++ API, the UObject Java API has two limitations: it is available only to create remote UObjects and remote Java UObjects can only run on computers having the full Urbi native SDK installed, since the Java library is generated from the C++ SDK implementation, and rely on compiled C++ code. In Aliz-e these limitations are not a problem since the UObjects run on classic operating systems where Urbi SDK is completely available.

3.3 Urbi App for NAO

The porting of Urbi for NAO is available on the Aldebaran Store at:

`https://cloud.aldebaran-robotics.com/application/urbi` (Aldebaran account required for access).

It consists of an Urbi engine cross compiled for NAO, several adapter UObjects to interact with Aldebaran NAO QiMessaging middleware, and urbiscript code to provide an abstraction layer over all of the software modules provided by Aldebaran.

4 Installation

This section is a step-by-step guide to install the complete ALIZ-E system. You will find the sources for the system in our subversion source repository, located at `https://svn.alize.gostai.com/trunk/`

4.1 Material Required

To install the system, the following equipment is required:

- a NAO robot
- an Ubuntu 12.04 computer to run the system
- a tablet to play the Quiz activity
- a touchscreen table to play the Sandray activity
- an extra screen to play the Dance activity

4.2 Initial Robot Setup

- Install Naoqi software with version 1.22.1 on a NAO Next Gen robot (`http://www.aldebaran.com/en/humanoid-robot/nao-robot`).
- Setup the network and log your robot on the Aldebaran cloud
- Install the urbi3 application from the Aldebaran Store.
  `https://cloud.aldebaran-robotics.com/application/urbi/`
- If you intend to use Dance behaviour, update urbi using this patched version:
  `https://cloud.aldebaran-robotics.com/application/urbi/5244/download/urbi4qimessaging-atom-12`
  `zip`
- Go to the NAO local web-page (`http://nao.local/apps/urbi`), start urbi and enable auto-start
- Run 'visudo' in root and add the following line:

```
nao     ALL=NOPASSWD:/home/nao/alize/bin/initial-setup.sh
```

- Get `https:/cloud.aldebaran-robotics.com/application/urbi/5244/download/urbi4qimessaging-atom` `zip` and extract the content into /.local/share/PackageManager/apps/urbi/urbi4qimessaging
- Open `/home/nao/.local/share/PackageManager/apps/urbi/urbi4qimessaging/share/gostai/nao.u` and insert at line 518:

```
load("/home/nao/alize/share/urbi/util/load-on-nao-new.u");
```

  and insert at line 80:

```
memory.declareEvent("footContactChanged");
```

4.3 Initial Computer Setup

4.3.1 Requirements

Here is the list of packages needed in Ubuntu 12.04:

```
sudo apt-get install automake autoconf libtool g++ cmake subversion
 default-jdk default-jdk-doc openjdk-7-jdk libgstreamer0.10-dev
 libgstreamer-plugins-base0.10-dev gstreamer0.10-x
 gstreamer0.10-plugins-base gstreamer0.10-plugins-good
 gstreamer0.10-plugins-ugly gstreamer-tools python sox audacity rlwrap
 cvs flex libsndfile-dev libasound2-dev ant libavutil-dev
 libavcodec-dev libavformat-dev libswscale-dev libavc1394-dev
 libv4l-dev libtiff4-dev mysql-server mysql-client libmysql-java
```

The system must be configured to use Java 7:

```
sudo update-java-alternatives -s java-1.7.0-openjdk-i386 # (on a 32 bits system)
sudo update-java-alternatives -s java-1.7.0-openjdk-amd64 # (on a 64 bits system)
```

4.3.2 ALIZE Checkout

Checkout the ALIZ-E source code using subversion:

```
svn co https://svn.alize.gostai.com
```

The ALIZ-E source code is in the `trunk` folder. From now on this folder will be referred to as $ALIZE_DIR

4.3.3 Urbi 3

Download Urbi 3 here :

`https://cloud.aldebaran-robotics.com/application/urbi/5244/download/urbi-suite-1221-linux64tar.gz`

or for a 32 bit system:

`https://cloud.aldebaran-robotics.com/application/urbi/5244/download/urbi-suite-1221-linux32tar.gz`)

Uncompress it in any place and make sure to set $URBI_ROOT environment variable to the location of this directory. Also make sure $URBI_ROOT/bin is in the $PATH environment variable, or the system will complain that it does not find `urbi-launch`.

These variables should probably be defined in the shell start-up file (`~/.zshrc` for zsh, or `~/.bashrc` for bash).

Additionally an alias could be set up in the shell start-up file to launch urbi easily:

```
alias urbi='urbi-launch -s --'
```

4.3.4 MARY TTS

The supported version for ALIZ-E is 4.3.1 so download the following installer: `http://mary.dfki.de/download/4.3.1/openmary-standalone-install-4.3.1.jar`

MARY does not require a specific install directory. However, please note that there should not be any space characters in the path to MARY as it causes some errors (the default name proposed by the installer is therefore not good). The directory will be called $MARY_DIR henceforth.

Please be sure to install at least:

- istc-lucia-hsmm it female hmm
- cmu-slt-hsmm en_US female hmm (and also the en_GB language)
- bits1-hsmm de female hmm

And do not install MBROLA voices (this generates an error).

Some pronunciations are corrected with the new component for Italian, a new voice has been trained, and a patch for fixing audio burst on synthesis time has been designed. To install this patch, do:

```
cd \$ALIZE\_DIR/manuals/marytts_patches/
./extract_mary_patch.sh \$MARY\_DIR
```

Then run the MARY TTS server:

```
\$MARY\_DIR/bin/maryserver
```

4.3.5 Julius ASR

You can compile and install julius4 with proper options by directly downloading sources from CVS, giving the following commands:

```
cd \$ALIZE\_DIR
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.jp:/cvsroot/julius co julius4
cd julius4
CFLAGS=-fPIC ./configure --prefix=\$(pwd)
make
make install
cd ..
```

4.4 Compile, Install and Deploy ALIZ-E

4.4.1 Compilation

As previously mentioned, an important requirement that has changed from the previous version of the system is that the Java JDK version 7. The system will not compile with Java 6 or earlier. From $ALIZE_DIR, you can use `./build.sh` to compile the whole system and install it in `__build/install` To do this manually, here are the commands that are run:

```
mkdir __build
cd __build
cmake -DCMAKE_INSTALL_PREFIX=\$(pwd)/install ..
make
make install
```

In the following instructions, `$ALIZE_DIR/__build/install` will be called $ALIZE_INSTALL.

Troubleshooting

- If cmake fails, look at the error. The most common problems are:
  - You are missing some external requirement (Glib2-dev, gstreamer-dev, etc.)
  - You did not checkout and/or compile Julius, or you put it in the wrong place
  - Your URBI_ROOT is not set (or not to the right value)
  - You try to use an old version of Urbi (<3.0)
- If make fails, this is almost always because of a missing external requirement (not all of them are checked by cmake)
  - if building afe fails with an error such as:

    ```
    make[3]: *** No targets specified and no makefile found. Stop.
    make[2]: *** [afe_build] Error 2
    make[1]: *** [components/afe/CMakeFiles/afe_build.dir/all] Error 2
    make: *** [all] Error 2
    ```

    Go to the components/afe directory and run: `autoreconf -i` Remove the `__build` directory, and retry.
  - If you get Java compilation errors (about JComboBox and other interface things) it is very likely because you are using Java 6. The new integrated system requires Java 7. Install the Java 7 JDK, check your system is using it (with java -version) and try again.

4.4.2 Deploy on the Robot

Go to $ALIZE_INSTALL and run:

```
bin/sync_robot-new.sh <Nao IP>
```

Once it has run, restart Urbi (or naoQi, or the complete Nao). To restart urbi and naoQi in a reliable way you can use `bin/restart-remote-urbi.sh <Nao IP>`

Troubleshooting

- If the end of the sync script fails and displays a message about the duties of a server admin, it means you forgot to add the visudo line; do it and try again.
- If the robot says "I'm done" immediately after "Starting Urbi", it means that Urbi crashed when it tried to start. Try again; if the problem persists, that may be caused by a stuck gstreamer instance on NAO. To solve it, ssh into the nao and run : `pkill -9 'gst*'` and then restart NaoQi.
- If after starting urbi the robot says it has an error ("I just caught a fatal error, etc.") it means you are probably using load-on-nao.u and not load-on-nao-new.u Check your file `/home/nao/.local/share/PackageManager/apps/urbi/urbi4qimessaging/share/gostai/nao.u` on the robot.

4.4.3 Install the Dance Resource Server

The resource server has to be run on a computer with a display and sound that can be seen by the child during the dance. For testing purposes this can be the same computer as the one running the main ALIZ-E system.

To install the server on the computer <server IP>, go to `$ALIZE_INSTALL/share/resources/resource-server/` and run:

```
./install_resourceserver.sh <server IP> <your login on that computer>
```

If it doesn't work or you don't want to use that, you can just copy the archive `$ALIZE_INSTALL/share/resource-se` to the resource server PC, untar it, and run `./launch_server.sh <server IP>`

More detailed instructions regarding the resource server installation and use can be found in $ALIZE_INSTALL/share/resource-server/README

4.5 Run ALIZ-E

To run the system, go to $ALIZE_INSTALL and do:

```
IP_DISPLAY=<server IP> USER_DISPLAY=<your login on that computer> bin/start-remote-new.sh <Nao IP>
```

or for the fake (virtual) robot:

```
IP_DISPLAY=<server IP> USER_DISPLAY=<your login on that computer> bin/fake_robot-new.sh
```

(If you don't want to use Dance you don't have to set the *_DISPLAY environment variables)

If you changed and recompiled some robot-side code, use `sync_robot-new.sh` and then run the previous line. You can also use the deploy-new.sh script, which does the same thing.

99% of the time there is no need restart Urbi or NaoQi to run the system, even if changes have been made to the robot-side code. Urbi will only need to be restarted if:

- The system crashed and it can't be restarted by killing and restarting bin/start-remote-new.sh
- If changes have been made to the load-on-nao-new.u file

In those cases `bin/deploy-new.sh <Nao IP>` can be used. It will sync the robot, restart urbi and launch the system.

Troubleshooting

- If an error like "I just caught a fatal error, etc." occurs, then you probably forgot to start MARY.
- If the system seems to start but stops with the robot-side Urbi log showing something about the foot plate, you are encountering a rather annoying Urbi3 / naoqi 1.22 bug. The workaround is to start the system with the robot in a position where the bottom of its feet do not touch the ground. For example, you can have it sat with its feet forming an angle with the ground.

- If the startup process stops after the message "Connecting to the robot and waiting for ALIZ-E to be launchable..." there is probably a problem with the home/nao/.local/share/PackageManager/apps/urbi/
on the robot. Check that it loads load-on-nao-new.u (and not load-on-nao.u) and that the line is at the right place, the restart Urbi and retry.
- If the audio frontend remote dies unexpectedly ("Process Audio frontend has died") it may be because of a compilation error. Try the same fix as in 2a: go to <ALIZE dir>/components/afe, run "autoreconf -i", remove the `__build` directory, and rebuild the system.

5 Components

This section presents the different components used in the system.

5.1 Wizard GUI

The graphical user interface for the Wizard-of-Oz (WoZ), as seen in Figure 8, allows control of the behaviour of the robot during the experiments, and also the ability to:

- start and stop the different activities available on the robot
- switch from one activity to another
- manage small talk sessions

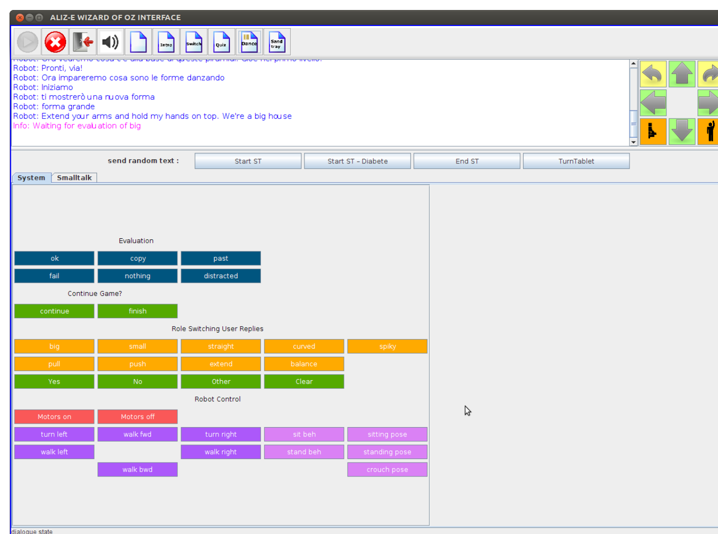

Figure 8: GUI for dance activity

The WoZ interface consists of the following elements:

- Control button bar: This is the topmost pane that contains buttons for starting or stopping the system, starting or stopping the various activities, and switching between completely supervised or semi-autonomous modes.

- Robot movement pane: This pane provides controls to move the robot, and make it stand up or sit down.

- Interaction transcript pane: Interaction transcript pane is the second topmost window in the interface. In this pane one can see the flow of the dialogue between the robot and the user.

- System actions tab: One of the bottom tabs where all possible system actions that could be done at the current point of the dialogue flow can be seen. At the very start of the system the pane is empty. When the operator starts a dialogue, buttons for all predefined dialogue steps appear here.

- User responses tab: Also at the bottom of the window, displaying buttons for possible user responses. At the start of the system the pane is empty. The buttons for possible user responses appear when the operator starts a new dialogue.

5.2 Activity Manager

Source code: `https://svn.alize.gostai.com/trunk/components/integration-core/`

The Activity Manager manages a list of activities the robot can run. It knows how to start, stop, pause, unpause and interact with them. See 7.1 for more information.

5.3 Level 0

Source code: `https://svn.alize.gostai.com/trunk/components/baseline-behavior/`

Description: A simple behaviour that is run when the robot is not playing with the child. It starts the BasicAwareness behaviour and the Perlin noise.

5.4 Basic Awareness

Source code: Basic awareness is a proprietary module found in NaoQi.

Description: ALBasicAwareness is a simple way to make the robot establish and keep eye contact with people. More documentation can be found online at `https://community.aldebaran-robotics.com/doc/2-1/naoqi/peopleperception/albasicawareness.html`

5.5 Dialogue Manager

Source code: `https://svn.alize.gostai.com/trunk/components/dialogue-flow/`

Description: The dialogue manager is responsible for controlling the robot's conversational behaviour. It is based on a hierarchy of reinforcement learning agents that learn their dialogue policies from a simulated environment (partially estimated from data). Such policies specify a mapping from dialogue states that describe situations in the interaction to actions roughly corresponding to dialogue acts. The communication of this component with the other components assuming Wizarded behaviour is as follows. The user says something to the robot (e.g. 'ask me a question') and the Wizard selects a dialogue act for the corresponding utterance such as 'Request(Question)'. Then the dialogue manager executes the given action by passing a dialogue act to the language generation and motor control components. Then the dialogue manager observes its new dialogue state, and suggests the next set of actions based on its dialogue policies and user simulations in the Wizard-of-Oz GUI. From this setting, if we remove the wizard then the system can behave autonomously, where user inputs (i.e. user dialogue acts and gesture acts) are derived from the language and gesture understanding components.

5.6 Motion Generation

Source code: `https://svn.alize.gostai.com/trunk/components/motion/`

Description: A simple behaviour that is run when the robot is not playing with the child. It starts the BasicAwareness behaviour and the Perlin noise.

The motion generation component, developed and maintained by UH, is the component responsible for the generation of movements, emotional displays, and synchronization with other components within the integrated system for all activities.

The main encapsulating class for the control and generation of movements is Motion Execution. It monitors the temperature of NAO's joints and changes their stiffness accordingly. If these joints belong to the lower limbs involved in standing and walking, the robot sits down. From this

interface class, one can perform movements, perform modulated movements, control (switch on and off) the Perlin noise, control the posture of the robot as well as the openness of the body, and trigger emotional displays.

5.6.1 NaoConfig

This class contains the initial configuration of the components. It is possible to set the joints on which the Perlin Noise can be applied to, and the initial amplitude and frequency applied to them. One can also choose an XML file path from which to load the predefined movements the robot can perform. These movements are loaded in a ScriptedMovement object at the time of instantiation of the MotionEx class.

5.6.2 Scripted Movements

This class first loads a list of movements (encoded as joint angles and times) from an XML file retrieved from the NaoConfig object. The movements are then stored and accessed through a hashmap linking a movement and a string. Movements include greetings, activity-based movements, head and gaze movements, and emotional key poses. Within the XML file, one can also design complex movements as a group of other movements, which allows the Motion Execution class to produce movements faster, having them pre-loaded and not generated in real time.

5.6.3 Perlin Noise

This class handles the Perlin Noise generation for entire body of the robot. Depending on what the robot is doing at a given time, it turns on or off the Perlin Noise on the joints that are not in use. Each joint has an instance of its own Perlin Noise (as a remote Urbi UObject), that can be modulated at will in terms of amplitude and frequency. These instances run in parallel. This upgrade leads to a more responsive system when the Perlin Noise is turned on and off in order to produce a given movement, or a sequence thereof.

5.6.4 Body Openness

This class modulates the posture of the robot using a real number bounded between 0 and 1. This is/can be interfaced to the Valence variable of the continuous emotional model, to reflect continuous and real time internal state of the robot.

5.6.5 Dance Movements Generation

This class generates trajectories of predefined dance movements, and uses predefined Choregraphe generated files.

5.7 Automatic Speech Recognition

Source code: `https://svn.alize.gostai.com/trunk/components/asr/`

Description: This component is responsible for automatic speech recognition. We are using the Open-Source Large Vocabulary CSR Engine Julius. See 7.8 for more information.

5.8 Face Emotion Recognition

Source code: `https://svn.alize.gostai.com/auxiliary-software/emoface/EmoFace_18_07_2014/`

Description: EmoFace is a windows-based command-prompt executable that by capturing frames from a camera at standard resolution can provide information on 2Dface position, 3D face pose,

and predict levels of arousal/valence (based on facial features). More information on this component can be found in Annex 7.5.

5.9 Kinect Recognition

Source code: `https://svn.alize.gostai.com/auxiliary-software/emoface/EmoFace_18_07_2014/`

Description: The purpose of this component is to record skeleton movements detected by a Kinect device. More information on this component can be found in Annex 7.6.

5.10 Memory System

Source code: `https://svn.alize.gostai.com/trunk/components/memorysystem/`

Description: Within ALIZ-E, the Memory System (as the outcome of WP1) is intended to form a bridge between the prior experience of the robot in interaction with human interactants, and present and ongoing behaviour. In brief summary, rather than forming explicit representations of experience in a symbolic/formal manner, the Memory System forms sub-symbolic cross-modal associations which subsequently act as the substrate for activation dynamics. The emphasis is thereby on the 'soft' coordination of multiple modalities through the mechanism of priming (itself inherently founded on prior experience), a feature that is necessary in the complex and dynamic task domain that is long-term social human-robot interaction. For further details please refer to D1.1, D1.2 and D1.4.

For a complete description of the API, please refer to Annex 7.7.

5.11 Motion & Speech Synchronizer

Source code: `https://svn.alize.gostai.com/trunk/components/syncout/`

Description: There is currently no simple way to synchronize robot movements and speech. It was agreed that it would be possible to at least start related movements and speech at the same moment. That's the purpose of this urbiscript component. It also supports queuing of speech and movement requests.

5.12 Text to Speech (TTS)

Source code: `https://svn.alize.gostai.com/trunk/components/tts/mary/`

Description: This component is responsible for the generation of speech from text. It relies on the Mary TTS Synthesiser. Mary TTS is a Text-to-Speech Synthesis platform written in Java using the client/server paradigm. Due to the NAO CPU resource limitations, it has been decided to run the Mary TTS server on a remote PC. When Mary TTS produces the audio stream, the resulting speech must be played on the NAO loudspeaker. This has been achieved using a streaming server based on gstreamer[1]. In order to have a real time interaction, a RTP (Real-time Transport Protocol) streaming server is active on NAO. The incoming audio RTP stream is then connected to the robot's loudspeakers. To bring Mary TTS and gstreamer RTP into the Urbi world, an Urbi Object (`UMaryTTS`) has been created as the principal Object responsible for routing the synthesis request (Mary TTS client) and for playing the resulting audio to different output channels. These channels are represented by the following Urbi Objects:

- `UMaryTTSAudioPlayer` makes a request to the Mary TTS server and plays the resulting synthesized voice through the PC loudspeakers (useful for the fakerobot simulation).

[1] `http://gstreamer.freedesktop.org/`

- `UMaryTTSAudioPlayer` makes a request to the MaryTTS server and streams the resulting synthesized audio through an RTP connection[2] using the efflux library[3].
- `UMaryTTSGstreamerPlayer` makes a request to the MaryTTS server and streams the resulting synthesized audio through a UDP RTP permanent connection using gstreamer-java[4].

The `Global.tts` API includes: `say(pText)`; `sayWithEmotion(pText, pEmotion)`; `isEmotionSupported()`; `setLanguage(pLanguage)`; `stopTalking()`. Moreover the following event is emitted two times in the synthesis phase with two different payloads: 1. `event_isSpeaking!(1)`: when the TTS has stared to speak; 2. `event_isSpeaking!(0)`: when the TTS has finished speaking.

More information on how to install this component can be found in Section 7.9.

5.13 User Model

Source code: `https://svn.alize.gostai.com/trunk/components/dialogue-flow/`

Description: The user model stores user data for future retrieval by the system components. More info on this component can be found in Annex 7.10.

5.14 Voice Activity Detection (VAD)

Source code: `https://svn.alize.gostai.com/trunk/components/vad/`

Description: The Voice Activity Detector (VAD) is an important element for facilitating the child-robot interaction. The VAD module allows the robot to detect that dynamically varying sound sources (such as human speech) which could be of interest for further analysis are active, in order to trigger further processing.

Our studies indicate that most VAD algorithms described in the literature are greatly affected by the type of background noise. Motivated by the fact that Nao will interact within an environment of unknown background noise conditions, we have examined the performance of different VAD algorithms for different background noise types that exhibit certain characteristics. In addition, a robust energy based voice activity detection algorithm has been proposed.

Within ALIZ-E, the VAD module implements a single-band energy based voice activity detector as a UObject C++ component. Each time an incoming audio segment is made available by the Audio Front End (AFE), the VAD processing function is triggered through the Urbi UNotifyChange mechanism. The VAD output is communicated to the listening modules via the Urbi event emission functionality: `event_voice_activity(payload, timestamp)`, where `payload = 1` when voice activity is detected, `payload = 0` when the voice activity has stopped and `timestamp` is used for logging.

The various VAD parameters (e.g. percentage of window overlap, Hangover time, Minimum pause length, etc.) are defined as UVars. This approach allows the easy modification and adjustment of these parameters according to the needs of a given use case or the noise conditions of the environment.

5.15 Voice Modification

Source code: `https://svn.alize.gostai.com/trunk/components/VoiceModification/`

Description: In order to support the Wizard-of-Oz experiments a voice modification module has been developed by VUB as a standalone UObject C++ component. This module enables the Wizard to speak through Nao in real-time, producing a modified (“robotized”) voice. The input signal (operator’s voice) which is made available through the AFE component is time-scaled

[2] This connection is created and destroyed every time a sentence is synthesized.
[3] https://github.com/brunodecarvalho/efflux
[4] http://code.google.com/p/gstreamer-java/

using the very robust WSOLA algorithm. The modified signal is then resampled to its original length. This operation results in shifting the original signal's pitch and thus creating a "robotized" voice effect. The produced signal is streamed and played though Nao loudspeakers using an RTP (Real-time Transport Protocol) streaming methodology based on gstreamer.

6 Cloud computing platform

Some of the ALIZ-E system components were implemented in a way which allows them to be hosted by a Cloud computing platform. For instance, one version of the Text-to-Speech component can be deployed in such a manner. Often the motivation behind this is that it is impractical to require consumers to have extra computers hosting robotic services on their local network (LAN) in a commercial robotic system. However, most of the ALIZ-E system components run in this manner not because of the commercial deployment advantages, but for the simplicity and distributed modularity that this approach offers. This section introduces Cloud computing hosting, and offers guidelines for implementation.

6.1 Cloud versus Embedded Services

First, it is important to note that deploying services on the robot or in the Cloud have major consequences for the implementation of the service, and what it is able to do. In the table below we summarize these differences.

Onboard	Hosted in Cloud
Direct access to robot services / devices	No access to robot services and devices
Fast communication with other robot services (< 50ms)	Slow communication with robot (> 300ms, up to seconds)
Small CPU consumption	High CPU consumption allowed
Small memory footprint	High memory footprint allowed
Limited on-board storage	Unlimited storage
Can be state-full	Preferably stateless
Real time access to robot HQ stream (video/audio/sensors)	Limited access to robot media stream (video/audio)
Available in case robot is disconnected from internet	Unavailable if no internet connection
Used by one robot	Used by several robots, possibly at the same time

Hosting a service in the Cloud is beneficial if a service requires a lot of CPU resources, or has a high memory footprint. The service can be used by a large number of robots, possibly accessing it at the same time. However this comes with some disadvantages:

- The service won't have direct access to the robot devices, or other embedded services.
- It will not have access to the full stream of data in the robot.
- It won't be able to react in real time to the robot state changes because of the limitation of network connections and bandwidth.
- It won't be available in case the robot does not have internet connection.

6.2 Guidelines

As part of the project, some guidelines for developing cloud robotic services were developed as follows:

- The service should be stateless if possible. This allows decoupling of the robot and the service; the service is not tied to a robot any more. It does not matter if the robot restarts or crashes. It can just access the service whenever it wants. In the same way, if the service crashes or restarts, there is no state to restore. If state information is required, then it should be stored by the robot, and sent to the remote service with every request the robot makes.
- The service should provide an HTTP API. This API should be Restfull. HTTP is a non-connected protocol, so the robot establishes connection on demand, whenever it needs to use the service. This negates the burden of maintaining long-lived connections between a robot and a server, especially because this is mostly impossible to maintain when it comes to unreliable networks using Wi-Fi. This also allows the service to be used by a greater number of robots. To further expand the number of robots using the service, the cloud platform can

host several versions of the same service on several virtual machine instances, and do load balancing between them. This is done easily with HTTP Restfull services.

With these guidelines, it is easy to create services such as:

- Text-to-Speech
- Automatic speech recognition
- Object recognition
- Memory repositories

HTTP Restfull services have some limitations: they can only be consumed by robots and they cannot push data to a robot. For most of services this is not a problem, and there are ways to overcome such problems. For instance, if a service is taking a lot of time processing before having the result ready for use by the robot, it is possible for the robot to poll the service API at regular intervals until it gets the result. This is not perfect, but will work in the majority of cases.

However, there are services that need to be able to contact the robot in real time. Remote tele-presence is such a scenario, where there is a user that wants to take control of the robot in real time, and must not have to wait. In such a case, it is possible to a use messaging middleware to pass messages to the robot. The NAO Robot uses the XMPP messaging protocol and can receive messages from a service that would require immediate contact with the robot to initiate a procedure.

Another limitation of HTTP Restfull services is that they cannot get a live media stream coming from the robots. For this particular purpose the service architecture must be different. The extended XMPP protocol currently available with the NAO allows creation of a media streaming session between a robot and a server. The SIP protocol can also be used to do the same thing (although not on NAO at the moment). While these technologies work well, the pitfall currently lies in the inability of networks to transport the media stream of the robot correctly. Indeed, live streams coming from a NAO are currently too big to be carried over a network, and need to be compressed on the robot-side before being sent to a remote service. This compression cannot be done on the software-side because of the huge CPU consumption it requires. Until the NAO incorporates dedicated compression hardware it won't be possible to stream from the robot to a Cloud server. Server-side, a streaming server able to multiplex the stream to various consuming services would be required. Once these technologies are in place, it will be possible to have Cloud services consuming a live stream from the robot.

6.3 Embedded Fall-back

Cloud hosted services are not available for robots not connected to the internet. However, robots should still be able to operate autonomously when offline. Therefore, robots require a basic version of all robotic services (text-to-speech, voice recognition, face recognition, etc) to remain usable without an internet connection. The services hosted in the Cloud should never be considered as mandatory for regular robot operation. Cloud services can provide some added value (better recognition for instance), but the robot should never be completely dependant on them.

7 Annexes

7.1 Activity Manager README

```
############################################
# ALIZE Integration Core / Activity Manager #
#           Basic documentation             #
#   25/07/2013 - a.coninx@imperial.ac.uk    #
############################################



What it does right now
----------------------
- It manages a list of activities the robot can run
- It knows how to start, stop, pause, unpause and interact with
  activities
- It allows to very easily start, stop, switch, etc. activities
- All this is mostly theoretical for now since it is only implemented
  with Dance


What it should do in the future
-------------------------------
- Implement sensitive learning (the events are ready but they are not
  handled yet by the activity manager)
- Receive commands from an activity selection GUI (or later an
  autonomous activity selection system)


Quick example :
---------------
Global.AM;
[01668739] ActivityManager_0x7feb2647ed08

Global.AM.list;
[01673227] ["quizz", "dance"]

Global.AM.start("quizz");
[01681854] *** Waiting for activity quizz to start...
[01681854] *** Blah blah i am currently starting a new quizz session
[01684857] *** Quizz is started, notifying the core
[01684859] *** Activity quizz started

Global.AM.start("dance");
[01691051] *** Activity quizz already running. Stop it if you want to start a new one.

Global.AM.pause_activity;
[01703559] *** Waiting for activity quizz to pause...
[01704561] *** Activity quizz paused

Global.AM.resume_activity;
[01711446] *** Waiting for activity quizz to resume execution...
[01712449] *** Activity quizz resumed

Global.AM.switch_activity("dance");
[01753272] *** Waiting for activity quizz to stop...
[01754274] *** Activity quizz stopped
[01754279] *** Waiting for activity dance to start...
[01754315] *** Configuring dance session...
[01754566] *** FSM(warmupShapes)::controlEvents
<snip>
[01754828] *** Activity dance started
[01754905] *** Hi! My name is Nao.
```

```
[01754941] *** Tell me, your name is

Global.AM.stop_activity;
[01810441] *** Waiting for activity dance to stop...
[01810614] *** ---- DanceManager::FSM -> leaving run loop.
[01810807] *** ---- DanceManager::stop -> Finished stopping dance beh
[01810849] *** Activity dance stopped

Global.AM.start("quizz");
[01876148] *** Waiting for activity quizz to start...
[01876148] *** Blah blah i am currently starting a new quizz session
[01879151] *** Quizz is started, notifying the core
[01879153] *** Activity quizz started

Global.AM.current_activity;
[01886257] "quizz"
```

Where is it now?

In the SVN trunk. Most of the code is in components/integration−core, and there is also the file /share/urbi/behaviours/integrated.u which is some kind of main.

How to test it

- To quickly test it on the fake robot just compile the project (cmake, make, make install) and run bin/launch_core.sh
- More generally, you have to load behaviours/integrated.u. This will check all that has to be launched is launched (especially remotes) and then create and setup the Global.AM object which is the activity manager

How to use it

- The example above covers everything that is implemented right now and is quite self explainatory.
- All calls are blocking until the relevant activity has been properly started/stopped/etc.

How to make my activity available in the activity manager

- You only need to write 2 small files and setup handlers for a few events
- Take example on the dance (fully implemented) and the fake quizz (just comments and code skeleton)

- Each activity is identified by an activity name, which is a string like "quizz" or "dance". In this readme i will use "XXX"

- load_XXX.u : this file should contain code that :
  * loads and initialize any genereic (non user−specific) code about activity XXX
  * setup handlers for the events (see later). That can be done directly in the file or in activity code loaded by the file.

- start_XXX.u : the code in this file should start the activity. This can be a simple line calling a "start" method but you may want to do some configuration before, for example by pulling data from the usermodel and using it to setup the ongoing session.

- The events are described and commented in details in the am−events.u file.
- To have your activity basically work with the AM, you should :

* Catch Global.AMevents.stop?("XXX"), and make it terminate the activity. It does not have to stop immediately, you can terminate nicely but you should terminate.
* Emit Global.AMevents.activityStarted!("XXX") whenever the activity has been successfully started.
* Emit Global.AMevents.activityStopped!("XXX") whenever the activity has been successfully stopped, just before terminating. It should emit it whatever and the way and the reason it has stopped, it may be as a consequence of catching Global.AMevents.stop?("XXX") or not.

- To have your activity implement pause and unpause, you should :
  * Catch Global.AMevents.pause?("XXX"), and make it pause the activity. It does not have to pause immediately but should do it in a reasonable time.
  * Emit Global.AMevents.activityPaused!("XXX") whenever the activity has been paused, either as a consequence of catching Global.AMevents.pause?("XXX") or not.
  * Catch Global.AMevents.resume?("XXX"), and make it resume the activity if it is paused.
  * Emit Global.AMevents.activityResumed!("XXX") whenever the activity has been resumed from paused, either as a consequence of catching Global.AMevents.resume?("XXX") or not.

- To have your activity implement sensitive listening (when it is be available) :
  * Catch Global.AMevents.requestSL?("XXX") and :
    + If the activity is in a state where it is possible to do some sensitive listening, put the activity in a state compatible with it (for example by preventing the use of speech) and emit Global.AMevents.readyForSL!("XXX")
    + If the activity can't do sensitive listening now, just emit Global.AMevents.denySL!("XXX")
  * Catch Global.AMevents.SLfinished?("XXX") and make the activity go on with its normal behavior after sensitive listening has been done.

- Finally you can add a line to make your activity known to the manager at the end of behaviours/integrated.u :

```
Global.AM.add_activity("XXX", "components/integration-core/load_XXX.u", "components/integration-core/start_XXX.u");
```

Creative Dance

Raquel Ros

July 23, 2013

1 Creative Dance Framework Design

Creative dance is a form of dance where the goal is to explore the body movement based on a set of general guidelines (movement concepts). Thus, on the one hand creative dance provides foundations on elements of dance, movement vocabulary and locomotive skills. And on the other, it promotes creativity motivating pupils to expand and extend movement range at their own rhythm through different stimuli, such as music, emotions, visual resources and observing the creations of others.

Moreover, creative dance can be used as an instrument to convey concepts that children work in their everyday school activities. Some theme examples are the states of water, creation and evolution of volcanoes, painting styles. Within the ALIZ-E context we focus the dance session on nutrition and healthy diet.

Each session covers the following stages:

1. Warmup: a sequence of movements is taught to be reproduced in the same way. In other words, the child copies the robot's movements.

   Each step (or movement) is shown one at a time to evaluate the performance of the pupils. Once a step is taught, it is added to the sequence. The expanded sequence is practiced every time a new step is added.

2. Exploration: it allows the introduction of the foundations of dance, i.e. movement concepts, along with the exploration of the body movement.

   The pupils go through a process of walking around the room and stopping to learn a new concept. The first time the concepts is described and performed while showing an image of it in the screen. Next, they are asked to walk around the room again and stop to show another example of the same concept. The robot waits for the child to show something. If it fails (either because the motion is incorrect or because it does not do anything), then the robot shows another example. This process is repeated until all the concepts are taught.

   Once they have gone through the different concepts, the robot creates a sequence of concepts and asks the child to reproduce them while giving counts. It first waits for the child to create a motion on its own. If it fails (meaning that the motion is incorrect or the child does not move) then the robot shows an example. They repeat this process several times to go through all the concepts learned.

3. Linking knowledge: the selected theme (in this case, nutrition and healthy diet) is linked to the dance concepts learned. The robot first explains ideas about the theme and then links them with the movements seen so far.

   To consolidate the knowledge, it asks the pupils to reproduce movements after images of food presented in the screen.

2 Implementation

In the current version the concepts –both from dance and nutrition– are taught through three sessions.

2.1 Day 1

During the first day the children are taught about SHAPES. These are then linked to VEGETABLES and FRUITS.

1. Warmup: a fixed sequence of five movements is taught. Once all the movements are taught, the sequence is repeated a couple of times with music.

2. Exploration: five shapes are explored.

   - big
   - small
   - straight
   - curved
   - spiky

3. Linking: the robot explains the following while a set of images are shown in the screen:

   > "Vegetables and fruits provide vitamins and minerals to your body important to prevent from diseases. You should eat five a day, any combination you want, but at least five!."

   The robot then shows images of fruits and vegetables in the screen and asks the child to tell to which shape does the fruit or vegetable correspond to. For instance, a banana corresponds to a curved shape.

   Finally, the robot tells the name of a shape, and the child should try to reproduce with its body a fruit or vegetable with that shape. For instance, for spiky shape, the child could reproduce a pineapple.

Day 1 finishes.

2.2 Day 2

Children are taught about QUICK and SMOOTH ACTIONS. These are then linked to SIMPLE and COMPLEX CARBOHYDRATES respectively. Since two concepts are described in this session, the dance stages –exploration and linking– are repeated twice.

1. Warmup: the robots creates a sequence based on the shapes seen in day 1. The child should copy the moves.

2. Exploration: three quick actions are explored.

   - fidget
   - sudden
   - spring

3. Linking: the robot explains the following while a set of images are shown in the screen:

   > "Simple carbohydrates give us energy in a fast way. But in the same way, this energy goes away quick. You suddenly feel very active when you take it. But as soon as it is over, you drop! It's like a fake energy. It's better to avoid them!
   > Now, quick actions correspond to simple carbohydrates. The movements are sudden, quick, fast. Just like the simple carbohydrates act in our body. For instance, sweets, fizzy drinks, pastries."

   Finally, the robot shows images of simple carbohydrates and the child should name the types of actions they can do after eating those.

4. Exploration: three smooth actions are explored.

   - reach
   - balance
   - extension

5. Linking: the robot explains the following while a set of images are shown in the screen:

   > "Complex carbohydrates give as long and sustained energy. The energy lasts longer in our bodies and it won't fall suddenly. On the contrary, they provide the necessary fuel to our bodies. They keep us satisfied for longer. You can eat as many as you want!
   > So, smooth actions correspond to complex carbohydrates. The movements are slow, smooth, sustained. The same way complex carbohydrates give us long energy. For example, rice, bread, pasta."

   Finally, the robot shows images of complex carbohydrates and the child should name the types of actions they can do after eating those.

Day 2 finishes.

2.3 Day 3

Children are taught about STRONG ACTIONS. These are then linked to PROTEINS.

1. Warmup: the robots creates a sequence based on the shapes seen in day 1 and the quick and smooth actions seen in day 2. The child should copy the moves.

2. Exploration: five strong actions are explored.

   - push
   - pull
   - stamp
   - kick
   - swing

3. Linking: the robot explains the following while a set of images are shown in the screen:

   > "Proteins are the building blocks of our body. We need them to grow. They represent the strength. You can eat them, but in a controlled way. Too many is not good.
   > Now, strong actions correspond to proteins. The motion is strong, with power. Proteins act in the same way. For instance, meat, beans, fish, eggs, nuts."

   Finally, the robot shows images of proteins and the child should name the types of actions they can do after eating those.

Day 3 finishes.

7.3 Creative Dance README

```
Compilation and installation
−−−−−−−−−−−−−−−−−−−−−−−−−−−−
It’s all urbi based. Therefore, compile and install the overall system as
usual.


Running
−−−−−−−
Dance makes use of images. A server to display images and play music has
been developed in urbi.

1. Run the server:

   cd TRUNK_PATH/components/creative−dance/ipc_server
   urbi ipc_server.u IP_DISPLAY

   where IP_DISPLAY corresponds to the IP of the machine where the images
will be displayed. You can use localhost (for the same machine where you’re
running it)

2. Run the alize system (fake or real).

In the current state (24/07/2013) the new dance is not yet making use of
the general GUI. In the meantime, you can use an interface written in Gostai
Lab to wizard the interaction:

a. Open Gostai Lab.

b. Open the interface in TRUNK_PATH/components/creative−dance/woz.ula

c. Open a terminal to connect to the robot (fake or real):

   rlwrap nc localhost/robotIP 54000

d. Configure the dance session. Some parameters are automatically taken
from the usermodel (id, name, language, firstEncounter). So mark them in the
usermodel GUI. The others have to be created. The serverIP corresponds to
the IP_DISPLAY used above:

   usermodel.switchActivity("Dance");
   var session = 1;
   var availableTime = 20min;
   var firstDanceSession = true;
   var serverIP = "10.0.0.104";

   Global.dance.config(usermodel.getId,
                  usermodel.getGivenName,
                  usermodel.getLanguage,
                  session,
                  availableTime,
                  usermodel.getEncounter == "first",
                  firstDanceSession,
                  serverIP);

e. Start the game:

   Global.dance.start(),


NOTE:
−−−−
To better understand the dance game, please refer to the document
creative−dance.pdf in TRUNK_PATH/manuals/.
```

7.4 Creative Dance Server README

What is the resource server ?
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
It is the piece of code responsible for displaying images on a screen and playing music during the dance activity (It is also capable of playing videos, though this is not used at the current time). The source is in share/ resource−server It is written in Urbi and can be controled using a TCP socket and a simple text−based protocol.

Where should it run ?
−−−−−−−−−−−−−−−−−−−−−
It should run on a computer connected to a screen and speaker in the room where the child and robot are.
Which means, except for some testing purposes, NOT the same system as the Wizard of Oz / Simon / GUI / whatever you call it.

Requirements ?
−−−−−−−−−−−−−−
− Reasonably modern linux system (MacOS can work with some tweaking. Windows won’t due to limitations in URBI for Windows.)
− A working X server
− Urbi >= 2.7.3
− Programs feh and vlc (packaged in debian an ubuntu)
− Automatic installation & remote starting also require an ssh server, md5sum, screen, tar and gunzip

In the following explanations:
− nao_IP is the IP address of NAO which is the robot
− server_IP is the IP address of SERVER_PC the computer on which you want to run the resource server (usually a computer with speakers and a screen in the experimental room)
− WoZ_IP is the IP address of ALIZE_PC which is the main computer running the ALIZE GUI, the remotes, etc.

As I wrote, SERVER_PC and ALIZE_PC can (technically speaking) be the same machine, but you won’t want that in a real experimental setup (unless you want to deal with a complicated dual−screen setup).

How do I install/run/start it correctly ?
−−−
Two ways :
− The automated way :
∗ On ALIZE_PC go to install/share/resources/resource−server and run :
./install_resourceserver.sh <server_IP> <user login>
with the IP of the computer you want to use as the server and you user login on this computer. It will check the environment, the current server version (if any) and install/update it if needed. Afterwards the server will be installed in ∼/alize−resource−server/resource_server (You will have to type in your password plenty of times, this is normal, use SSH public key login if you want to avoid that).
∗ After the server is installed, on ALIZE_PC run:
./launch_remote_server.sh <server_IP> <user login>
to automatically launch the server on the remote host. It will be launched in a detached screen named " resourceserver", run screen −r resourceserver on SERVER_PC to monitor it

− The manual way :
∗ Copy the install/share/resources/resource−server/resource_server.tar.gz on the computer you want to use as a server and untar it
∗ cd to the resource_server dir and run urbi−launch −s −− resource_server.u <server_IP>
(The server IP should be ∗the own IP of the computer you are running the server on, on the interface it will use to communicate with the robot∗. Not WoZ_IP or nao_IP.)

How do I make the system / the dance game use it ?
−−
It still relies on the IP_DISPLAY environment variable, but I updated the starting scripts (especially launch_remote_uobjects.sh) and the dance code so it is now much more convenient :

- If you just don't want to use the resource server... don't do anything. Just start the system using bin/fake_robot.sh or bin/deploy.sh <nao IP>. Dance should work and simply not show any image nor play any music.

- If you want to use the resource server, set the IP_DISPLAY environment variable to server_IP. The system will check that the server is running when starting. If it is, good. If it is not, it will display a big flashy warning in the log file saying that dance will not work, and go on loading the system.

- If you set both the IP_DISPLAY environment variable to the server_IP USER_DISPLAY variable to your user login on SERVER_PC, the system will attempt to remotely launch the server at start if it is not running. This will only work if the server is installed in ~/alize−resource−server/resource_server

FULL EXAMPLE
−−−−−−−−−−−−
With the automatic installation system:
****
On ALIZE_PC:
cd <ALIZE install dir>/share/resources/resource−server
install_resourceserver.sh <server_IP> <your login in SERVER_PC>
 (enter your passphrase/password when asked, let it install...)
cd <ALIZE install dir>
IP_DISPLAY=<server_IP> USER_DISPLAY=<your login in SERVER_PC> bin/fake_robot.sh
 (or)
IP_DISPLAY=<server_IP> USER_DISPLAY=<your login in SERVER_PC> bin/deploy.sh <nao_IP>

(If you run the system multiple times you can omit the USER_DISPLAY variable after the first run since the server is already started)

Without the automatic installation system:
****
On ALIZE_PC:
cd <ALIZE install dir>/share/resources/resource−server
scp resource_server.tar.gz <your login in SERVER_PC>@<server_IP>:
 (or transfer it using an USB key, e−mail, etc.)

On SERVER_PC:
 (Copy the resource_server.tar.gz wherever you want)
tar xzf resource_server.tar.gz
cd resource_server
urbi resource_server.u <server_IP>
 (check it starts correctly)

On ALIZE_PC:
cd <ALIZE install dir>
IP_DISPLAY=<server_IP> bin/fake_robot.sh
 (or)
IP_DISPLAY=<server_IP> bin/deploy.sh <nao_IP>

Don't hesitate to ask me if you need any further information or help !

−−
Alexandre Coninx
a.coninx@imperial.ac.uk

|------0------|

EmoFace

0 0

|

-_-

face emotion recognition for affective computing

18_07_2014 (Version 1.4)

ETRO-VUB

EmoFace is a windows-based command-prompt executable that by capturing frames from a camera at standard resolution (640x480)can provide information on 2Dface position, 3D face pose, and predict levels of arousal/valence (based on facial features). Tested on a 2GHz laptop, running 64-bit win8, a frame rate of 12 to 14 frames was achieved with the wireless Ai-Ball camera.

The module can send events to the NAO robot. The event format and sample code on how to capture these events can be found on Alize SVN:

components/vision-events-listener/vision-events.u

components/vision-events-listener/vision-events-handler.u

The module can work with Windows (64-bit, version 7 or 8) and can read video streams from:

1) Laptop internal camera
2) External USB camera
3) Ai-Ball wireless camera
4) A video file (OpenCV supported formats)

5) NAO's internal camera

and performs the following video processing tasks:

1) Face detection and face 2D position in the frame
2) 3D pose of the head (roll, pitch and yaw angles)
3) Facial based Arousal/Valence prediction

Due to its small size and weight (only 100gr), Ai-Ball can be used as an alternative to NAO's internal/head camera, providing higher frame rate at VGA resolution, necessary for better accuracy. When NAO is on Wi-Fi, its head camera can send maximum 2 fps (@640x480). With the Ai-Ball wireless camera we achieved 12 to 14 fps (@640x480) with the video processing running in Matlab. The wireless camera can be fitted easily on top of NAO's head, with a double-sided tape. Ai-Ball can be found @Amazon.uk (delivery only in UK).

Platform:

Can work with **64-bit Windows 7 or 8 machines.**

Installation:

Step 1:

Install the following runtimes:

- Visual C++ Redistributable for Visual Studio 2012 Update 4 (**vcredist_x64.exe**)
- MATLAB Compiler Runtime 8.2 (**MCRInstaller.exe**)

Both installation files are included in previous **EmoFace** release (EmoFace_06_02_2014).

Step 2:

Copy the folder **EmoFace** at a destination on a PC, running 64-bit Windows 7 or 8.

Set AI-Ball camera:

Follow the instructions of the user manual and configure the camera in "Infrastructure LAN Mode". Do not set USER NAME and PASSWORD (leave these spaces blank!!). Keep the IP address, so you can use it later (as IP_CAM).

Running the application:

Step 1:

Modify **the Param.txt** file, according to your preferences. There are 6 fields that need to be used:

INPUT_ID 0 → Laptop internal camera

INPUT_ID 1 → External USB camera

INPUT_ID 2 → Ai-Ball wireless camera

INPUT_ID 3 → A video file (OpenCV supported formats)

INPUT_ID 4 → NAO's internal camera

IP_NAO xx.xx.xx.xx → IP address of NAO, when events will be send

IP_NAO FALSE → If NAO is not used

VISUALIZATION TRUE → If the video image will be viewed

VISUALIZATION FALSE → If the video image will not be viewed

VIS_POINTS TRUE→ Tracking points will appear on the video

VIS_POINTS FALSE→ Tracking points will not appear on video

VIDEO_FILE videofile.extension → If a video file will be viewed and processed (supported extensions from OpenCV)

VIDEO_FILE FALSE→ If another video stream source will be used

IP_ AIBALL xx.xx.xx.xx → IP address of Ai-Ball wireless camera

IP_ AIBALL FALSE → If Ai-Ball wireless camera is not used

For example, if you use Ai-Ball with IP address 192.168.0.150, with visualization and willing to send events to NAO with an IP 192.168.0.100, the Param.txt file should be like:

INPUT_ID 4

IP_NAO 192.168.0.100

VISUALIZATION TRUE

VIS_POINTS TRUE

VIDEO_FILE FALSE

IP_AIBALL 192.168.0.150

Step 2:

Save the Param.txt file and close it.

Step 3:

In command prompt (cmd) switch to the folder **EmoFace_18_07_2014** and run **EmoFace.exe**. When the 3 steps of initialization are done, then tracking and sending of events to NAO can be started by selecting OK on **Start Tracking** window. To terminate the process, select OK on the **Stop Tracking** window.

Using NAO's internal camera:

With EmoFace it is possible to use also NAO's internal camera. Please follow the following steps in order to establish connection with NAO through a wireless router:

1) Switch on NAO, activate the Wi-Fi interface and figure out the Wi-Fi IP address
2) Modify NAO's IP Wi-Fi address in the nao_face_server_v5.py (line 23)
3) Put the same address in the parameters file of EmoFace and activate the NAO camera with TRUE;
4) Copy nao_face_server_v5.py to NAO using an FTP application (such as WinSCP on Windows);
5) Start the face image sever by typing 'python nao_face_server_v5.py' using a SSH Telnet terminal application (such as PuTTY on Windows);
6) launch EmoFace: the server on NAO detects your face (each time R is null), crops an extended face rectangle from the image, compresses it to a (160,160) image, and sends the image and metadata to EmoFace via TCPIP. The client, EmoFace, unpacks the image, tracks the face, and sends the face rectangle R to the server via TCPIP. Each time R is null, the server will stay in a loop until detects a face.

Structure of Events:

Events are send to NAO as a sequence of numbers with description:

"location" = X,Y [position of the head in the image]

"arousal" = number between -1 and 1 [arousal prediction]

"valence" = number between -1 and 1 [valence prediction]

"faceAngles" = X, Y, Z [rotation around X (X axis = head's center to left ear) , rotation around Y (Y axis = upwards), and rotation around Z (Z axis = towards the viewer)]

Version History:

```
28/01/2014 - Version 1.0 --> reads Ai-Ball, USB cameras, sends events to URBI:
face 2D / 3D position, arousal/valence

05/02/2014 - Version 1.1 --> added: 1)reading of video files 2)replace "NaN"
with "nan"

18/02/2014 - Version 1.2 --> added: 1) user selection for tracking points to
appear or not

24/03/2014 - Version 1.3 (BETA) --> added: 1) using NAO's internal camera

18/07/2014 - Version 1.4 --> various improvements for valence/arousal
prediction.
```

7.6 Kinect recognition README

```
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

| Kinect ALIZE component − install guide − 24/03/2014 |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+


I) Data collection and logging
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
There is data about the Kinect uobject in three places:
* The skeleton data is recorded in timestamped CSV files in data_collection/kinect
* Some high−level data about the kinect recording (when it is started, stopped, recording, etc.) is stored on the
      robot in alize/logs/<timestamp>_kinect−events.log It can also be used to achieve synchronization between
       the robot time and the kinect timestamps.
* The remote's debug output is in data_collection/logs/<timestamp>/kinect.log (It is only debug information,
      only useful to fix problems)


II) How it is behaving now in the integrated system
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

The general idea is that if there is a kinect, we use it, and if there is not, we don't complain and don't break
      anything:

* When starting the system, the remotes manager will try to load the UKinect remote. If it has been built it will
      do so, otherwise you will just have an error message "Remote Kinect NiTE2 interface died" and the system
      will just work without the kinect.
* When loading the activity manager, the system will check if the kinect remote (uobjects.UKinect) is here, and
      if it is it will instantiate it and try to initialize it. If it works, it will bind the initialized UKinect object to
      Global.kinect. If it fails (becasue there is no kinect plugged or because of a problem), it will just say it and
      *not* create Global.kinect. More information can be found in data_collection/logs/<timestamp>/kinect.
      log
* If it exists a Global.kinect object, event handlers will react on some activity manager events and drive the
      kinect:
  − The kinect skeleton tracking will start (and a new data file will be created) when you start a sesssion and
      stopped when you stop a session
  − Recording itself is started when you start the dance activity and stopped when the dance becomes stopped



III) Kinect UObject API
−−−−−−−−−−−−−−−−−−−−−−

When the remote is loaded, it provides the uobjects.UKinect class, which can be instantiated by the default
      constructor:

var kinect = uobjects.UKinect();

There is no reason to instantiate more than one instance of that class (the only one would be multiple kinects
      but it is not supported yet), and doing so is probably a bad idea.

Quick reference:


Execution control:
kinect.init_kinect(); : initializes both NiTE and the kinect itself. Returns false on failure (either it could not be
      initialized or it is already initialized), true on a success
kinect.start_kinect(); : starts the kinect motion tracking, which means the device is trying to detect users and
      follow them. Returns false on failure (if it is already running, or not initialized, or somehow could not be
      started), true on a success
kinect.start_recording(); : makes the kinect record the current skeleton data. A new data file is created and data
      will automatically be written when somone is tracked. Returns false on failure (if it is already recording, or
      not started), true on a success.
```

kinect.stop_recording(); : makes the kinect uobject close the current data file stop recording data. Returns false on failure (if data recording was already off), true on a success.
kinect.stop_kinect(); : stops the motion tracking. Also calls stop_recording implicitely if it was recording. Returns false on failure (if it is not started), true on a success.

Remote information:
kinect.isRunning(); : returns true if the kinect tracking is enabled, false otherwise
kinect.getTrackingStatus(); : returns true if the kinect is actually tracking someone right now, false otherwise
kinect.isRecording(); : returns true if the skeleton data recording is currently enabled, false otherwise

Runtime skeleton data access:
In all those functions, name is a string describing a NiTE2 joint name, like "JOINT_HEAD" (look in the NiTE2 doc or in ukinect.cpp for the list)
kinect.getJointPosition(var name) : returns a vector of 3 floats describing the joint position (x, y, z). Returns (0, 0, 0) if no information is available
kinect.getJointPositionCertainty(var name) : returns a float describing the certitude of NiTE2 about the position of that joint. 1 is dead certain, 0 is completely unknown. Returns 0 if no information is available.
kinect.getJointRotation(var name) : returns a vector of 4 floats describing the joint orientation as a quaternion (w, x, y, z). Returns (0, 0, 0, 0) if no information is available
kinect.getJointRotationCertainty(var name) : returns a float describing the certitude of NiTE2 about the orientation of that joint. 1 is dead certain, 0 is completely unknown. Returns 0 if no information is available .

——
Alex
a.coninx@imperial.ac.uk

Technical Report (working document)
The ALIZ-E Distributed Memory System (DAIM) API

PLYM (PEB)

August 2014

Abstract

This technical report describes ALIZ-E Distributed Memory System API, including updates made up to the end of the project. While the Memory System itself has been subject to continual changes, the API has only been extended, and has remained backward compatible. The main assumptions made in the implementation of this API are described. For details regarding the theoretical operation of the Memory System itself, please refer to the relevant publications and documentation. All source code is present on the ALIZ-E SVN.

Contents

1 Outline and Assumptions **2**

2 Walk-through example **2**

3 API Reference **7**

3.1 Class: ModInt 7

3.2 Class: SimpleMod 9

4 Issues and Extensions **10**

4.1 Known issues 10

4.2 Possible changes/extensions 10

5 Glossary **11**

6 Changelog **11**

1 Outline and Assumptions

The intention of this tech report is not to describe the mechanisms of operation of the ALIZ-E Memory System, but rather to describe how it can be interfaced with and used in the wider cognitive architecture. This API relies on the fulfilment of a number of assumptions based on theoretical/conceptual issues (described elsewhere), which have a number of practical consequences. It should be noted that the Memory System itself is subject to ongoing development, however, based on the aforementioned assumptions, it is envisaged that the core API will be subject to very little change, with only potential extensions to this core (e.g. see section 4). It is always the intention to maintain backward compatibility.

The big change in this version of the API is that the Memory System will now run in 'Active Mode' by default. This means that every predefined interval, the Memory System will request new activation information from each of the Modalities registered to it, and will subsequently return an activation profile to each of these modalities. Previously, a Memory update cycle would only be initiated if a modality requested an update. This 'Passive Mode' is still present in the Memory System, and may be reactivated if desired.

The assumptions regarding the interface and operation of the Memory System with all Cognitive Modalities have been described and discussed previously[1]. They are briefly as follows:

1. That **discrete units of processing are distinguishable within each of the cognitive modalities** linked to the Memory System. In the context of the Memory System, these discrete processing units are termed "(cognitive modality) Objects".
2. That these **Objects are persistent for the duration of run-time**. This does not preclude the addition of new objects during run-time (although this functionality is not yet included in the API).
3. That these **Objects have an equivalent functional role within the given cognitive modality**. For example, within a modality where there are hierarchies of objects, the objects that may be interfaced with the Memory System must be drawn from the same level within the hierarchy.
4. That these **Objects have a property intrinsically involved in ongoing processing within that modality that may be regarded as an activation level**. This could correspond to such properties as probability of presence, degree of involvement, etc. In the context of the Memory System, activation is a floating point number (actually a double) in the range [0.0,1.0], where 1.0 denotes maximal activation. Having this property linked to the processing within the cognitive modality is important, as this is the means by which the Memory System can influence processing based on prior experience.

2 Walk-through example

In order to illustrate the main functionality of the memory system, a passive mode operation example will be described: i.e. a modality external to the memory system determines when it performs an update. The normal mode for the memory system is now active: each modality runs at its own update rate, and the memory system periodically and independently runs its update cycle. This enables the memory system to run in real-time, which was not possible in passive mode.

Note also that this example differs significantly in terms of setup and execution from the manner in which it is run in the context of the ALIZ-E integrated system. This example is focussed on how the memory system can be interfaced with, and not how it is used in the integrated system.

[1] See presentations made in Viareggio (Sept 2010), Saarbrucken (March 2011), Padova (June 2011), Paris (Sept 2011), and Amsterdam (Nov 2011) for outline of assumptions and structures envisaged for use.

The following example will go through the setup, initialisation and use of the Memory System by three example (very simplistic) cognitive modalities, in an Urbi environment[2]. Examples have already been placed on the SVN - see particularly loadMemory.u, initMemory.u and api-test.u. *The initialisation step in the following code is not central to the working of the Memory System API, but is performed here so as to provide example modalities.*

First load the MemorySystem uobject and classes in urbi, assuming that the uobject has already been compiled. The Memory System UObject may be run in either remote or plugin mode. This example assumes that all classes and the Memory System uobject are in the same directory, and loads the Memory System UObject in plugin mode.

SETUP: see loadMemory.u for a full example

set the path, let's pretend it is correct....

$var\ \ PATH$ some_name;

first the uobject...

$loadModule(PATH +$ /memorysystem.so);

then the classes...

$loadFile(PATH/$MemIntData.u);
$loadFile(PATH +$ /ModInt.u);

and finally the example cognitive modality...

$loadFile(PATH +$ /SimpleMod.u);

The necessary classes and modules have now been loaded, and the initialisation of the various objects can be start. Loading the "ModInt" source file will automatically create an object called MemoryInterface - it will also automatically create an instance of the Memory System UObject. This is assumed in the following example, where two Modalities are created (colour and shape) each with two objects.

INITIALISATION: see api-test.u for a full example

create new modality instances

$var\ \ colour = SimpleMod.new("colour");$
$var\ \ shape = SimpleMod.new("shape");$
$var\ \ label = SimpleMod.new("label");$

Create some sample object identifiers for these modalities in this example two objects to be assigned

$var\ \ objs_colour = ["red", "blue"];$
$var\ \ objs_shape = ["square", "circle"];$
$var\ \ objs_label = ["blibble", "wooble"];$

Add to modalities: NOTE just for the purposes of this example

$colour.add_objects(objs_colour);$
$shape.add_objects(objs_shape);$
$label.add_objects(objs_label);$

Register modalities with the ModInt object. See Reference section, SimpleMod for description.

$colour.setup_interface;$
$shape.setup_interface;$

[2]NOTE: all of these classes and code examples have been tested on an Urbi 2.7.5 server running on Ubuntu 12.04, 32-bit.

$label.setup_interface;$

Then setup the Memory System itself using the registered modalities MUST ENSURE that all modalities have been registered before performing this step

$var setup_complete = MemoryInterface.setup_interfaces;$

Simple check to see if the Memory System has been set up correctly

```
if(setup_complete) echo("1"); //setup complete
else    echo("0"); //setup failed
```

The Memory System has now been setup and initialised, having been linked to the three sample modalities with identifiers colour, shape and label. What remains to do is to start the Memory System update to request activation from these modalities: 'Active Mode' of the Memory System. In order to demonstrate this, in the following example, the SimpleMod instantiations are assigned an activation level to simulate incoming data/ongoing processing. See the reference section for more details. A Memory-initiated update cycle consists of the following:

(1) memory (in ModInt) detects need for new update cycle;
(2) this triggers an event, which is used to request activation updates from all linked modalities (in this example: colour, shape and label);
(3) the activation is passed to the Memory System uobject which performs a Memory System update cycle;
(4) an event is triggered to notify the modalities that updated activation is available;
(5) each modality retrieves its updated activation profile and processes as required/desired;
(6) end of update cycle; (1) triggers after period p_{time}.

In the following example, the memory system update rate is set to 500ms (p_{time}). The API function Global.MemoryInterface.start, is used to start processin,; and the API function Global.MemoryInterface.stop; is used to stop memory processing.

RUNNING: see api-test.u for a full example

```
var count_a = 0;
```

Function to initiate an update cycle

```
function start_cycle()
{
```

Initiate cycle

```
    count_a++;
```

Go through some of the sample data
Get the next modality activation data

```
    colour.next_input;
    shape.next_input;
```

Send modality activation to ModInt structures

```
    colour.activation_to_memory;
```

Cycle triggers a memory update

```
}
```

Setup event triggering every 200ms to start new cycle ensure updates don't overlap

```
{
```

start_cycle;

Add break, try for 3 iterations

if (count_a == 3)
{
break;
}
}

If the example files are used, then on each of the three time-steps, information will be dumped to terminal showing the activation levels of each modality object at each stage of the update cycle (i.e. before and after the Memory System update cycle). The following information is dumped to terminal for three time-steps of the above example (which corresponds to the presentation of a blue circle for three time-steps).

```
=> loadFile("/home/pebaxter/Dropbox/PROGRAMMING/Urbi/MemorySystem
    /Memory System UObject v1/urbi/loadMemory.u");
#[0002384477] *** Start load of MemorySystem components:
#[0002384477] *** ---> memory system uobject loaded...
#[0002384479] *** ---> MemIntData class loaded...
#[0002384487] *** ---> ModInt class loaded...
#[0002384487] *** << ModInt >> Empty test complete...
#[0002384487] *** ---> tests completed...
#[0002384492] *** ---> SimpleMod class loaded...
#[0002384492] *** ***********************
#[0002384492] *** Memory System loaded...
#[0002384492] *** -> MemoryInterface
#[0002384492] *** ***********************
=> loadFile("/home/pebaxter/Dropbox/PROGRAMMING/Urbi/MemorySystem
    /Memory System UObject v1/urbi/api-test.u");
#[0002388764] ***
#[0002388766] *** **** Start of MemorySystem API simulation ****
#[0002388766] ***
#[0002388766] *** creating modalities and objects...
#[0002388767] *** add objects to modalities...
#[0002388767] *** register modalities with the ModInt...
#[0002388769] *** setup the memorysystem...
#[0002390772] *** setup of demonstration complete - start running...
#[0002390772] ***
#[0002390772] *** *=*=*=*=*=*=* start modint update 1...
#[0002390772] ***
#[0002390773] *** colour modality sending activation to memorysystem...
#[0002390773] *** [0.05, 0.9]
#[0002390773] *** ----- colour activation from Modality -----
#[0002390774] *** shape modality sending activation to memorysystem...
#[0002390774] *** .......to memory complete
#[0002390774] *** [0.05, 0.9]
#[0002390775] *** ----- shape activation from Modality -----
#[0002390775] *** .......to memory complete
#[0002390775] *** << ModInt >> start memory system update step...
#[0002390776] *** << ModInt >> -> activation data correctly constructed:
#[0002390776] *** [[0.05, 0.9], [0.05, 0.9]]
#[0002390777] *** << ModInt >> deconstructing received activation information:
#[0002390777] *** [[-0.0802, 0.2564], [-0.0802, 0.2564]]
```

```
#[0002390777] *** +++++ colour activation from MS +++++
#[0002390777] *** +++++ shape activation from MS +++++
---#[0002390778] *** colour modality retrieving activation from memorysystem...
#[0002390778] *** shape modality retrieving activation from memorysystem...
#[0002390779] *** << ModInt >> ...completed memory system update step
#[0002390780] *** [-0.0802, 0.2564]
#[0002390780] *** [-0.0802, 0.2564]
#[0002390780] *** ....... from memory complete
---#[0002390780] *** ....... from memory complete
#[0002390972] ***
#[0002390973] *** *=*=*=*=*=*=* start modint update 2...
#[0002390973] ***
#[0002390973] *** colour modality sending activation to memorysystem...
#[0002390974] *** [0.05, 0.9]
#[0002390974] *** ----- colour activation from Modality -----
#[0002390974] *** shape modality sending activation to memorysystem...
#[0002390975] *** .......to memory complete
#[0002390975] *** [0.05, 0.9]
#[0002390975] *** ----- shape activation from Modality -----
#[0002390975] *** .......to memory complete
#[0002390975] *** << ModInt >> start memory system update step...
#[0002390976] *** << ModInt >> -> activation data correctly constructed:
#[0002390976] *** [[0.05, 0.9], [0.05, 0.9]]
#[0002390977] *** << ModInt >> deconstructing received activation information:
#[0002390977] *** [[-0.0645184, 0.453171], [-0.0645184, 0.453171]]
#[0002390977] *** +++++ colour activation from MS +++++
#[0002390978] *** +++++ shape activation from MS +++++
#[0002390978] *** colour modality retrieving activation from memorysystem...
#[0002390978] *** shape modality retrieving activation from memorysystem...
#[0002390979] *** << ModInt >> ...completed memory system update step
#[0002390980] *** [-0.0645184, 0.453171]
#[0002390980] *** [-0.0645184, 0.453171]
#[0002390980] *** ....... from memory complete
#[0002390980] *** ....... from memory complete
#[0002391173] ***
#[0002391173] *** *=*=*=*=*=*=* start modint update 3...
#[0002391173] ***
#[0002391174] *** colour modality sending activation to memorysystem...
#[0002391174] *** [0.05, 0.9]
#[0002391174] *** ----- colour activation from Modality -----
#[0002391174] *** shape modality sending activation to memorysystem...
#[0002391174] *** .......to memory complete
#[0002391175] *** [0.05, 0.9]
#[0002391175] *** ----- shape activation from Modality -----
#[0002391175] *** .......to memory complete
#[0002391175] *** << ModInt >> start memory system update step...
#[0002391176] *** << ModInt >> -> activation data correctly constructed:
#[0002391176] *** [[0.05, 0.9], [0.05, 0.9]]
#[0002391176] *** << ModInt >> deconstructing received activation information:
#[0002391176] *** [[-0.0520986, 0.549485], [-0.0520986, 0.549485]]
#[0002391176] *** +++++ colour activation from MS +++++
#[0002391177] *** +++++ shape activation from MS +++++
```

```
#[0002391177] *** colour modality retrieving activation from memorysystem...
#[0002391177] *** shape modality retrieving activation from memorysystem...
#[0002391177] *** << ModInt >> ...completed memory system update step
#[0002391178] *** [-0.0520986, 0.549485]
#[0002391178] *** [-0.0520986, 0.549485]
#[0002391178] *** ....... from memory complete
#[0002391178] *** ....... from memory complete
```

Please note that due to the event-based nature of the update cycle, there is a possibility of overlaps in the notification of completion for some of the processes. In the above listing, this may be seen, for example, between timestamps [0002390778] and [0002390780] (indicated with three dashes) in the first update cycle, where activation profiles are retrieved from the Memory System for the two Modalities.

3 API Reference

There is only one main interface class that needs to be considered when interfacing a cognitive modality with the Memory System: ModInt (contraction of Modality-Interface). All interaction with the Memory System is handled using this urbi class. The example SimpleMod class is also described here to demonstrate how any given cognitive modality can register itself and its objects, and send and receive activation to/from the Memory System.

Not all methods for the two classes are listed here, only those that form part of the API. Additionally, the methods indicated with the symbol (†) are core parts of the API (as described in section 1). Examples for use can be found in section 2.

3.1 Class: ModInt

3.1.1 Events

- Three global events are defined during object initialisation and are used for the control flow of the Memory System.
- *Global._event_Mod_ModInt_complete*: triggers when all modalities have updated their activations with ModInt, in turn triggers a Memory System uobject update cycle. This is a ModInt-specific event, no need to provide a handler for this within a modality.
- †*Global._event_MSupdate_request*: is triggered by the ModInt object to request an activation update from the registered Modalities - all modalities should provide a handler to deal with this request. See SimpleMod example below.
- †*Global._event_MSupdate_complete*: is triggered by ModInt when a Memory System uobject update cycle has been completed - all modalities should provide a handler to deal with this event, to retrieve an activation profile from ModInt. See SimpleMod example below.

3.1.2 Method: init()

- Initialises the class - standard urbi constructor. No arguments or return value.
- Example: *var newmodint = ModInt.new;*

3.1.3 Method: start()

- To start the operation of the component: currently not implemented - assumption that component will start when loaded.
- Example: *ModIntobject.start;*

3.1.4 Method: stop()

- To stop the operation of the component: currently not implemented.
- Example: *ModIntobject.stop;*

3.1.5 Method: test()

- Provides a test for correct initialisation, and for availability of run-time dependencies/resources: currently not implemented.
- Example: *ModIntobject.test;*

3.1.6 Method: shutdown()

- Closes data files and stops the MemorySystem. No arguments or return value.
- Example: *ModIntobject.shutdown;*

3.1.7 †Method: register_modality(name)

- Registers a modality with the ModInt structure, takes one string argument (unique modality identifier). Returns "false" if a modality with the matching identifier has already been registered.
- Example: *ModIntobject.register_modality("mod1");*

3.1.8 †Method: register_objects(name, objs)

- Registers the objects of an already registered modality with ModInt. Argument1: string identifier of the modality; argument2: list of strings, each string being the unique identifier of a modality object (unique for the modality, not globally). No return value.
- Example: *ModIntobject.register_objects("mod1",["obj1","obj2","obj3"]);*

3.1.9 †Method: setup_interfaces()

- Sets up the Memory System uobject with the registered modality information. No arguments - returns "false" if not all modalities are registered successfully with the uobject.
- **WARNING**: should only be called once all modalities and their objects have been registered with ModInt; if any modalities are registered with ModInt after this method is called, they will not be registered with the MemorySystem itself.
- Example: *ModIntobject.setup_interfaces();*

3.1.10 †Method: activation_to_memory(name, a-data)

- Sends activation data to be stored in a temporary container in ModInt before being sent to the Memory System uobject for an update cycle. Argument1: string identifier of the modality from which the activation comes; argument2: list of floats corresponding to the activation values of the modality objects. Returns "false" if no matching modality identifier is found.
- **NOTE**: it is assumed that the order of the objects in the modality - as presented in lists to ModInt - remains static during run-time. This assumption means that the activation levels may be assigned to the correct modality object tag in the Memory System.
- Example: *ModIntobject.activation_to_memory("mod1",[double1,double2,double3]);*

3.1.11 †Method: activation_from_memory(name)

- Retrieves activation data from a temporary container in ModInt after being taken from the Memory System uobject after an update cycle. Argument1: string identifier of the modality from which the activation comes. Returns list of floats corresponding to the activation values of the modality objects; returns "false" if no matching modality identifier is found.
- **NOTE**: it is assumed that the order of the objects in the modality - as presented in lists to ModInt - remains static during run-time. This assumption means that the activation levels may be assigned to the correct modality object from this returned list.
- Example: *var activation = ModIntobject.activation_from_memory("mod1");*

3.2 Class: SimpleMod

3.2.1 Events

- As described in ModInt, there are two global events that need to be handled by a cognitive modality
- *Global._event_MSupdate_request*: is triggered by the ModInt object to request an activation update from the registered Modalities.
- *Global._event_MSupdate_complete*: is triggered by ModInt when a Memory System uobject update cycle has been completed.

3.2.2 Method: init(name)

- Initialises the class - standard urbi constructor. One string argument which is the unique modality identifier; no return value.
- Example: *var newmodality = SimpleMod.new("mod1");*

3.2.3 †Method: setup_interface()

- Sets up the ModInt object ("MemoryInterface" by default) with modality and object identifiers to initialise. No arguments. Returns "true" if setup is successful, returns "false" otherwise.
- **NOTE**: assumes that the modality already has a list of object identifiers to use - in the example shown this is furnished by the property SimpleMod._objs
- Example: *newmodality.setup_interface;*

3.2.4 †Method: activation_to_memory()

- Sends activation to the relevant temporary container in ModInt prior to a Memory System update cycle. No arguments. Will return "false" if the modality has not yet been registered with the ModInt object. Should be called when the Global._event_MSupdate_request event is triggered.
- **NOTE**: assumes that the modality already has a list of object activations to use - and that this list has the same order as that used to register objects with ModInt. Also assumes that the ModInt object has been initialised as "MemoryInterface".
- Example: *newmodint.activation_to_memory;*

3.2.5 †Method: activation_from_memory()

- Retrieves activation from the relevant temporary container in ModInt after a Memory System update cycle, and stores in local list SimpleMod._activation_from_MS. No arguments. Will return "false" if the modality has not yet been registered with the ModInt object. Should be called when the Global._event_MSupdate_complete event is triggered.
- **NOTE**: assumes that this list has the same order as that used to register objects with ModInt. Also assumes that the ModInt object has been initialised as "MemoryInterface".
- Example: *newmodint.activation_from_memory;*

3.2.6 Property: _data

- Contains sample activation data used (for 10 time-steps) in the simulation example described in section 2. It has no other use regarding the required functionality of a modality interfacing with the Memory System.
- On each time-step, the next entry in this data structure is called using the SimpleMod.next_input() method.

4 Issues and Extensions

4.1 Known issues

- **Update rate**: On occasion, if the active mode update rate is too high, then the memory system will experience latency issues. This is possibly due to the overlapping events between update cycles. From previous versions this issue has been mostly resolved through the use of cycle completion flags and checks, however, certain problems still remain. An update rate of 0.4s is used in the ALIZ-E integrated system with no issues.

4.2 Possible changes/extensions

- **Affective modulation**: there is currently no formal means of affective modulation of memory in the API – it is not envisaged to alter the existing API, rather an addition that has an effect on the internal mechanisms of the Memory System. In principle this has been defined (see Appendix B of D1.4), but has not yet been added to the ALIZ-E integrated system.
- **Refactoring**: source code can always be made cleaner/more efficient.

5 Glossary

- **Activation**: The activation level of an object corresponds to such properties as probability of presence, intensity, frequency etc and thus not solely determined by the state of processing within a given modality but also affected by other functional systems and their processing. In the context of the Memory System, activation is a floating point number (actually a double) in the range [0.0,1.0], where 1.0 denotes maximal activation. This property forms the link between processing with individual cognitive modalities and the modulatory effects of the memory system based on prior experience.
- **Cognitive Modality**: A functional component of the cognitive architecture responsible for processing a specific type of information, or organising a particular aspect of the system's behavioural repertoire. Examples could include face recognition, dialogue planning, speech recognition, etc.
- **DAIM**: the "Distributed, Associative and Interactive Memory" system, the name applied to the distributed memory system in the ALIZ-E deliverables and publications.
- **ModInt**: The ModInt class (a contraction of Modality-Interface) handles all interaction between the cognitive modalities and the Memory System. It effectively embodies the Memory System API in urbiscript.
- **(Modality) Object**: these are discrete units of processing that are distinguishable within each of the cognitive modalities. They are assumed to have no dimensional relationship between them (within a modality), and are assumed to persist for the duration of system run-time.

6 Changelog

- v1.0, Aug 2014: numerous stability improvements made, improved error handling, more complete automated logging, and full capability of integration with the ALIZ-E integrated system (specifically the sandtray activity) has been completed. New functionality: save/reload of memory system data
- v0.4, Jan 2013: changes made to reflect updates to both Memory System and API. *Multiple modality handling* and *incorrect logging* bugs both resolved.
- v0.1, Feb 2012: first version of API documentation.

7.8 Speech Recognition README

```
JuliusSimple Urbi component

1. COMPILATION

How to locally compile the component:
−−−−−−−
$ cd your−alize−gostai−svn/trunk/components/asr/asr−julius4−urbi
$ ./compile−component.sh [YOUR_JULIUS4_PATH]


2. URBI LAUNCH

2.1 How to locally launch the component
−−−−−−−
You have to use two terminals

[first terminal:]
Start server with urbi class loaded:

−− URBI 3 VERSION −−
$ rlwrap urbi−launch −−start  −−host 0.0.0.0 −−port 54000 \
  −− −−interactive −f src/JuliusSimple.u

−− (OLD) URBI 2.x VERSION −−
$ rlwrap urbi −i −−host localhost −−port 54000  \
        −f src/JuliusSimple.u

[second terminal:]
urbi−launch the .so uobject
$ ASR_MODELS_BASE_PATH=../asr−julius4−models/ urbi−launch −r \
        target/lib/libasr−julius4−urbi.so        \
        −−                              \
        −−host localhost −−port 54000


2.2 How to launch the component from the integrated system:
−−−−−−−
$ cd your−alize−gostai−svn/trunk/___build/install

You have to use two terminals

[first terminal:]
Start server with urbi class loaded:

−− URBI 3 VERSION −−
$ rlwrap urbi−launch −−start  −−host 0.0.0.0 −−port 54000 \
  −− −−interactive −f share/urbi/components/asr/JuliusSimple.u

−− (OLD) URBI 2.x VERSION −−
$ rlwrap urbi −i −−host localhost −−port 54000 \
  −f share/urbi/components/asr/JuliusSimple.u


[second terminal:]
urbi−launch the .so uobject
$ ASR_MODELS_BASE_PATH=share/alize/components/asr/models/julius4 urbi−launch −r \
  lib/libasr−julius4−urbi.so \
  −− \
  −−host localhost −−port 54000


3. TEST THE COMPONENT

From server (interactive) side, play with the following functions:
```

```
// load a configuration (AM/LM models)
// the following is in english
[urbi−console] Global.asr_julius.loadconf("en", "adult_testdfa" );

[urbi−console] Global.asr_julius.start_detached();
// this sends the recognition detached in background

// if your microphone is properly configured
// with alsa, you should now be able to speak
// and have your voice recognized

[urbi−console] Global.asr_julius.stop();

// if you want to try different kind of input:
[urbi−console] Global.asr_julius.loadconf("en", "adult_testdfa" );

// load an audio file:
[urbi−console] Global.asr_julius.setInput("file", "filename.wav" );
// filename.wav should also match model samplerate (e.g., 16kHz).

// or, alternatively, use the default input audio interface (default
// for julius−urbi):
[urbi−console] Global.asr_julius.setInput("mic", "default" );

[urbi−console] Global.asr_julius.start_detached();
[urbi−console] Global.asr_julius.stop();


4 TODO
# add features:
# 1. input from AFE through URBI
# 2. load several language models
# ....



############
# URBI INTERFACE SMALL DOCUMENTATION:

function loadconf( var lang, var id )
−−
        MUTEX.LOCK
        if( lang == nil || lang == "" )
                lang = JuliusSimpleConf.default_lang();
        if( id == nil || id == "" )
                id = JuliusSimpleConf.default_model(lang);
        julius_simple_uobject.u_load_config_wrapper( lang, id );
        MUTEX.UNLOCK


function start_sequential()
−−
        MUTEX.LOCK
        julius_simple_uobject.u_open_julius_stream();
        MUTEX.UNLOCK


function start()
−−
        loadconf( nil, nil );
        start_detached();


function start_detached();
−−
        detach( {function start_sequential()} );
```

```
function stop()
--
        julius_simple_uobject.u_close_julius_stream();




############
# URBI / C++ INTERFACE SMALL DOCUMENTATION:

julius_simple_uobject.u_init_julius();
--
        dummy function, can be safely removed


julius_simple_uobject.u_load_config_wrapper();
--
        example of use:
        julius_simple_uobject.u_load_config_wrapper("it","adult_testdfa");
        julius_simple_uobject.u_load_config_wrapper("en","adult_testdfa");

        this function
        1) PARAMETERS: receives a LANG and an ID strings
        2) REQUIRE an environment variable to be set
        from those values it then builds a filename for julius
        conf.  finally calls
        julius_simple_uobject.u_load_config(); with that
        filename;

        it is worth noticing that the env var and parameters
        depend on the machine where the component runs.


julius_simple_uobject.u_load_config();
----------------------------------------------
        1) PARAMETER: receives a filename string

        if ( isStreamAlreadyOpen != 0 ) // *gs_istc*
                fprintf(stderr, "julius stream is already open\n");
                return 2;


julius_simple_uobject.u_free_config();
----------------------------------------------
        if ( isStreamAlreadyOpen != 0 ) // *gs_istc*
                fprintf(stderr, "julius stream is already open\n");
                return 2;

        if ( mRecog )
                j_recog_free( mRecog );  // jconf will be released inside this
                mRecog = NULL;
                mJconf = NULL;
        else
                fprintf(stderr, "it seems that mRecog was already NULL, no free performed\n");



julius_simple_uobject.u_open_julius_stream(),
----------------------------------------------
        if ( !mRecog ) // *gs_istc*
                fprintf(stderr, "cannot open julius stream: mRecog not initialized\n");
                return 1;

        if ( isStreamAlreadyOpen != 0 ) // *gs_istc*
                fprintf(stderr, "julius stream is already open\n");
```

```
        return 2;

    isStreamAlreadyOpen = 1;
    ...
```

julius_simple_uobject.u_close_julius_stream();

```
---------------------------------------------
    if ( ! mRecog )
        fprintf(stderr, "no mRecog was set, exiting\n");
        return 0;

    isStreamAlreadyOpen = 0;

    return j_close_stream(mRecog);
```

7.9 Text-to-Speech README

```
##################################
# marytts.readme.txt
# Author: Fabio Tesser
# Institution: CNR-ISTC, Padova - Italy
# Email: fabio.tesser@gmail.com
##################################

MARY TTS SERVER Installation
============================

In order to run this component you need to have as prerequisite a MARYTTS server up and running.

The supported version for alize is 4.3.1 so download the following installer:
http://mary.dfki.de/download/4.3.1/openmary-standalone-install-4.3.1.jar

Install MARY wherever you want, that directory will be called <MARY dir> in the following. Note that you
      should not include any space character in the path to MARY as it causes some error (the default name
      proposed by the installer is therefore not good).

Please be sure to install at least:
- istc-lucia-hsmm it female hmm
- cmu-slt-hsmm en_US female hmm (and also the en_GB language)
- bits1-hsmm de female hmm
And do not install MBROLA voices (this seems to generate an error).

Some pronunciations are corrected with the new component for Italian, a new voice has been trained, and a
      patch for fixing audio burst on sythesis time has been designed.
To install this:
cd <ALIZE dir>/manuals/marytts_patches/
./extract_mary_patch.sh <MARY dir>

Then run the server:
<your MARY directory>/bin/maryserver


GSTREAMER RTP UMaryTTS component.
=================================

Requires:
libgstudp.so http://packages.debian.org/lenny/i386/gstreamer0.10-plugins-good/download
libgstaudiotestsrc.so http://packages.debian.org/lenny/i386/gstreamer0.10-plugins-base/download

Console 0 (ssh on NAO robot) | start a Gstreamer RTP server (read http://alize.gostai.com/wiki/wp4/
      How_to_enableuse_Gstreamer_and_RTPUDP_plugins_on_Nao first):
GST_PLUGIN_PATH=/home/nao/alize/lib/gst-plugins~lenny2_i386/ gst-launch-0.10 -v udpsrc port=5000
       ! "application/x-rtp,media=(string)audio,clock-rate=(int)16000,width=16,height=16,encoding-name=(
      string)L16,encoding-params=(string)1,channels=(int)1,channel-positions=(int)1,payload=(int)96" !
      rtpL16depay ! audioconvert ! alsasink

# Other option:
gst-launch-0.10 -v udpsrc port=5000 ! "application/x-rtp,media=(string)audio,clock-rate=(int)16000,width
      =16,height=16,encoding-name=(string)L16,encoding-params=(string)1,channels=(int)1,channel-positions
      =(int)1,payload=(int)96" ! rtpL16depay ! audioconvert ! autoaudiosink

# with explicit buffer size:
GST_PLUGIN_PATH=/home/nao/alize/lib/gst-plugins~lenny2_i386/ gst-launch-0.10 -v udpsrc port=5000
       buffer-size=16384 ! "application/x-rtp,media=(string)audio,clock-rate=(int)16000,width=16,height=16,
      encoding-name=(string)L16,encoding-params=(string)1,channels=(int)1,channel-positions=(int)1,
      payload=(int)96" ! rtpL16depay ! audioconvert ! alsasink

Console 1 | launch maryserver (i.e.:):
       cd $MARYPATH
       bin/maryserver

Console 2 | launch urbiserver:
```

```
rlwrap urbi −i −H localhost −P 54000
```

Console 3 | launch remote component:

```
cd $SVN_ALIZE/components/tts/mary
./start−component.sh $(readlink −f ../../../share/config/config.sh)
```

Console 2 | create a new object and test on urbi:

```
var tts=UMaryTTSRTPPlayer.new;
tts.start;
tts.say("hello!");
```

If you want to observe the events emitted by the tts, use something like this:

```
at (tts.event_isSpeaking?(var isSpeaking)) {
    echo("tts started speaking event with isSpeaking="+isSpeaking);
};
```

IMPORTANT: There are a number of settings relevant for MARY TTS, including the RTP server host, which can be set in ../../../share/config/config.sh.

7.10 User Model README

This describes the Usermodel for the Alize−project.

Geneal description
==================

The classes:
UUsermodel.java: UObject for the Usermodel
Usermodels.java: Main container of all usermodels
Usermodel.java: A model of an individual user. It implements GeneralData and uses SpecificData.
GeneralData.java: Class to contain the data of the user like name, age etc.
SpecificData.java: Class to contain the data specific to an activity.
UsermodelGUI.java: GUI to monitor and change the data and contains some Wizard of Oz functionalities.
SpecificDataPanel.java: Class to create the GUI of the data specific to an activity.
GoalServerSocket.java: Classes to connect to Goal
SocketListener.java:

General idea:
There is a model for each user. One of the models contains the data for the NAO.
GeneralData contains more or less static data like "Name", "is diabetic"

The data for the activity is stored in its SpecificData.

The Usermodel is started from launch_remote_uobjects.sh which loads the usermodel.jar.
This load the UObject UUsermodel. Its start function calls the constructor of the Usermodels.
In the share/urbi/behaviours/main.u the UUsermodel is instantiated. That has to be done before the other components are started since they (propably) are going to use that usermodel.

General Data
============

In the file "share/resources/config/UsermodelData.xml" the following properties can be set:
− the path to save the datafiles
− the flag if the GUI is to be shown
− the default Id
− the default Activity
− the names of the activities
− the languages can be set.

There is also a tag debug. With that four buttons are visible to perform tests. They are not usefull in normal circunstances.

The GeneralData contains the following properties:
String Id
int Age
String Diabetes flag true if child has diabetes
String Debut flag, "true" if child has diabetes for less than 2 years and he is less than 9 years old
int Experience number of years since the child has diabetes.
int Activationlevel data from memory

String FamiliarityCond flag if the child is familiar
String FrameOfReference if the child's right is right
String Language
String FamilyName
String GivenName
String Gender

int EmotionalState
int AttentionState
int ArousalState
int ValenceState

In the UObject UUsermodel each of these properties has a getter and setter function.

Activity specific data

======================

The parameters of an activity are defined in the file UsermodelData.xml (in the share/config/ folder)
With the definition also the type, visibility (on the GUI) and default value can be set.
Type supported are: Integer, String, Double, Bool, ArraylistofString, ArraylistofInteger and ArraylistofDouble.
Examples:

<default name="ArraylistofString" type="ArraylistofString">[one, two three]</default>
<default name="ArraylistofInteger" type="ArraylistofInteger">[1,2,3]</default>
<default name="ArraylistofDouble" type="ArraylistofDouble">[4.1, 5.2]</default>
<default name="empty" type="ArraylistofInteger"></default>
<default name="emptyAlso" type="ArraylistofInteger">[]</default>

Due to Urbi implementation any bool value is converted into an Integer of 0 (false) and 1 (true)

During runtime also parameters can be stored in the SpecificData section. That data will be visible.

If there is a data files in the resource folder that will be used.
But if the configurationfile contains parameters which are not in the saved file then that parameter will not be added.

Application interface
=====================

With the following functions in UUsermodel the properties can also be changed:
setGeneralDataValue(UDictionary)

Or the data from the activities:
Note: First one needs to select the activity!
setActivityDataValues(UDictionary data)
setActivityData(String key, UValue data)
Where data can be of type:
Integer
Double
String
UList (listitems being of type Integer, Double or String)

To retreive data from the activities one can use:
var data = getActivityData("key") (selects from current active activity)
var data = getSpecificActivityData("activity", "key")
var data = getActivityDataValues() (returns dictionary with all parameters of the current activity

If the data value is a string containing "true" of "false" the parameter will be set the corresponding boolean.

NOTE!!!
Given these strange things I am still considering to change the code so only doubles and strings are supported!

setEmotionalDataValue(UDictionary)
with the key of the dictionary being the name of the property to set.

Whenever properties are changed with these functions a notifyEvent is emitted containing a dictionary with the actual values:
event_notifyGeneralDataChanged
event_notifyActivityDataChanged
event_notifyEmotionalDataChanged

Other functions in UUsermodel:
switchID
switchActivity

Data from the usermodel is being persisted in files. The location is "share/resources/usermodel/"
And the names: "id"_GeneralData.xml, "id"_DanceData.xml, "id"_QuizData.xml, "id"_ImitationData.xml, "id"_MathData.xml.
Also a history is kept of changes during the session. Those are persisted into filenames containg the name history
In the GUI tabs are shown for GeneralData and the actual defined activities.
The tabcolor of the active activity is reddish (when it is not selected)

On these tabs the properties are depicted. Which can be changed.
The buttons beneath the tabpages are Wizard of Oz buttons
− to change the level of the emotional states.
− to indicate if the child should recieve feedback (positive, neutral or negative)

With the close button the GUI closes.

When the system is started
− there is a slot called usermodel.
− all available usermodels are loaded

While editing the field will become white. When the data is excepted the field will become grey again and show the stores data.

− empty integer will be set to 0
− empty double will be set to 0
− empty list will be set to []
− when entering data from code there are some "issues"
    single values like integer, double and string:
        a string can be converted into integer or double
        an integer can be converted into double and visa versa
        an integer or double can nit be converted into a string
        can de converted into any list value
    list values:
        can be converted to and from integer, double or string
        can be converted into single integer or double. string will stay list of string

If the types in the list are different
    − if any numerical value is a double the list will be of type double
    − if any item in the list is non−numerical the loist will be of type string
    − there is no posibility to discriminate between a UValue of 1 or of 1.0 both are treated as double
    − so a list will be of type double if at least one item of the list has a decimal value not 0

Urbi examples:
==============

usermodel.switchId(5); => uses user number 5

var user = ["GivenName" => "TestName", "FamilyName" => "User"];
usermodel.setGeneralDataValue(user); => set name of the user

var user1 = ["Age" => 12, "Gender" => "female", "Language" => "italian"];
usermodel.setGeneralDataValue(user2); => set age, gender, language.

usermodel.setGender("male");
usermodel.setDiabetes("true");

usermodel.getAge(); returns the age of the user.
usermodel.getFamilyName(); returns the name.

usermodel.switchActivity("Dance"); => the Dance−tab is shown.

var dictar = ["a" =>"a","c" => 1.2, "b" =>"d", "d" => [1.2, 1.3], "e" => ["one", "two"]];
usermodel.setActivityDataValues(dictar);
var ar = usermodel.getActivityData("e"); => ar = ["one", "two"]

var strar = ["a","c","b","d"];
usermodel.setActivityData("trystr", strar);
var newstrar = usermodel.getActivityData("trystr");

var doublear = [1.5, 1.75];
usermodel.setActivityStrFloatDictValue("trydouble", doublear);
var newdoublear = usermodel.getActivityData("trydouble");

To test the implementation a TextUsermodel.u is available.

Standalone installation
======================

Run Usermodel outside the Alize framework

. Set up an Urbi server:
$ rlwrap urbi −H localhost −P 54000 −i
(alternatively you can run naoqi with urbi enabled, please refer to Aldebaran SDK for more information).

2. From a different shell launch the Usermodel:
$ cd Your/trunk/components/usermodel/StandAlone
$ urbi−launch−java lib/usermodel.jar −−classpath=lib/kxml2−2.3.0.jar:lib/xstream−1.4.2.jar

3. From a third shell connect to the urbi server:
$ rlwrap nc localhost 54000
Start Usermodel with:
$ if (!Global.hasLocalSlot("usermodel")) var Global.usermodel|;
$ Global.usermodel = UUsermodels.new|;
$ usermodel.start|;

7.11 Voice Activity Detection README

```
--------------------------------------------------
| ALIZ-E Project - Voice Activity Detector (VAD)|
--------------------------------------------------
Author: Giorgos Athanasopoulos
Email: gathanas@etro.vub.ac.be
Last update: 19.09.2012

REQUIREMENTS:
g++ 4.4.x
URBI SDK 2.7.1
Note: the VAD has been tested in Ubuntu 10.10, 32bit.


*************
A. COMPILING:

        The following environment variables need to be set (e.g. in
        $HOME/.bashrc or $HOME/.profile):

        export URBI_ROOT=/YourPathHere/urbi-sdk-2.7.1-linux-x86-gcc4
        export PATH=$URBI_ROOT/bin:$PATH

        You may compile the VAD by:
        $ g++ -Wall -I $URBI_ROOT/include -fPIC -shared *.cpp *.c -o \
        uVAD.so

        For compiling the VAD on mac os x 10.6:
        $ cd ~/YourVADPath
        $ g++ -Wall -I $URBI_ROOT/include -fPIC -shared \
        -Wl,-undefined,dynamic_lookup *.cpp *.c -o uVAD.so


*************
B. INSTALLATION:

        1. Set up an Urbi server:
        $ rlwrap urbi -H localhost -P 54000 -i
        (alternatively you can run naoqi with urbi enabled, please
        refer to Aldebaran SDK for more information).

        2. From a different shell launch the VAD:
        $ cd ~/YourVADPath
        $ urbi-launch -r uVAD.so -- --host localhost --port 54000 &

        3. From a third shell connect to the urbi server:
        $ rlwrap nc localhost 54000
        From here you can send commands to the urbi server and the
        VAD. The commands and interfaces are covered in section C and
        D.


*************
C. RUNNING THE VAD:

// Create VAD component (assuming that the Audio Front End has been
started):

        var Global.vad = uVAD.new(Global.afe.getSlot("val"));

Please refer to the AFE documentation for more information on the AFE.

// Set the VAD parameters, otherwise the default ones (same as the
ones indicated below) are used. More details on these parameters are
given is section E.
```

```
        vad.u_fs = 16000;
        vad.u_windowLength = 32e-3;
        vad.u_windowOverlap = 0.50;
        vad.u_initLength = 0.100;
        vad.u_initialSoundPercentage = 100;
        vad.u_frontHangoverTime = 0.150;
        vad.u_hangoverTime = 0.200;
        vad.u_minPauseLength = 0.300;
        vad.u_minSpeechLength = 0.250;
        vad.u_nrUsedAdjacentFrames = 6;
        vad.u_threshold = 3;
        vad.u_varThreshold = 0;
        vad.u_threshSNRpercentage = 50;
        vad.u_usedFreqBand0 = 300;
        vad.u_usedFreqBand1 = 6000;
        vad.u_minSpeechPowerdB = -40;
        vad.u_alphan = 0.99;
        vad .u_alphap = 0.999;

// Set the VAD recorder parameters, otherwise the default ones (same
// as the ones indicated below) are used. More details on these
// parameters are given is section E.

        vad.u_filepath = "./";
        vad.recorder_timeout_speech = 8;
        vad.recorder_timeout_no_speech = 8;
        vad.u_minAudioPortion = 0.350;
        vad.multiple_recording = "false";

// Start the VAD module. This method starts the VAD with the
// parameters defined above.

        vad.start();

        For changing a (or more) parameter, first stop the VAD (see
        stop method), change the value of the parameter and start
        again the VAD.

// Stop the VAD module:
        vad.stop();

// Check the status of the VAD module. This method can be used for
// debugging, i.e. to check whether in an integrated system the VAD is
// still running or not. If the VAD is running, the used parameters are
// displayed.

        vad.ping();

// A recording can be started by setting the UVar below.
        vad.do_recording = "start";
        Please refer to section D for more information on the specifications of the audio file that is produced.
        A recording starts when speech activity is detected and is terminated when:
        a) the speech activity is finished and the speech segment lenght is longer than the u_minAudioPortion
       parameter.
        b) there is no sufficient speech detected for a period equal to the recorder_timeout_no_speech parameter.
       In this case no file is produced.
        c) there is continuous speech activity for a period equal to the recorder_timeout_speech parameter.
        After a recording is finished, the do_recording UVar is set to "idle".

// A recording can be stopped (before the previous conditions are met) by setting the following UVar:
        vad.do_recording = "stop";
        If a recording is stopped,
        a) a file is written if the recording contains sufficient speech activity (defined via u_minAudioPortion
       parameter);
        b) no file is written is the recording contains no speech activity.
        After a recording is stopped, the do_recording UVar is set to "idle".
```

// A recording can be cancelled (before the previous conditions are met) by setting the following UVar:
vad.do_recording = "cancel";
If a recording is cancelled, no file is written and the do_recording UVar is set to "idle".

// Initiating multiple recordings of audio segments
In order to create multiple recordings, each of the containing the detected speech segments, the following parameter needs to be set:
vad.multiple_recording = "true";
(by default this parameter is set to "false")
The recording can be started as before (vad.do_recording = "start";). The VAD after completing the recording of the first detected segment will reset the do_recording from "idle" to "start", so that when the next segment containing speech is detected a new file will be created. This will continue until the parameter " multiple_recording" is set to "false", or the "do_recording" is set to "stop" or "cancel".

// Display the VAD output (e.g. whenever it is updated) can be done by checking the value of the output UVar, e.g.:

```
Global.vad.&output.notifyChange(uobjects_handle, closure() {
 echo ("VAD output: " + vad.output);
});
```

Please see section D for the values that that the output may take.

// The delay introduced by the VAD, depends on the choice of parameters and can be checked via the u_delay UVar:

```
echo ("VAD total delay (in samples): " + vad.u_delay);
```

****
D: VAD INTERFACES (EVENTS)

The current implementation assumes that the VAD receives periodically audio data from the Audio Front End interface (via the "vad.input" UVar). The voice activity detection is therefore continuously running. The VAD provides its output (on whether speech activity is detected) through the "vad.output" UVar.

The VAD output is either −1, 0 or 1. During the initialisation (which introduces a delay with length that depends on the choice of parameters) the output will be −1. After that the output will be 0 if the frame was classified as a pause frame and 1 if it is a speech frame.

Whenever there is speech activity detected (transition of the VAD output from 0 to 1) or the speech activity stops (transition for the VAD output from 0 to 1), the event_voice_activity event is emitted.
The event_voice_activity has as argument "1" when speech activity is detected and "0" when the speech activity stops, e.g.:

```
at(vad.event_voice_activity?(var status, var timelog)) {
    echo("vad sent voice activity status="+status);
    },
```

The VAD will record the incoming audio and notify the ASR module when a new recording is available. It is expected that the DM module will trigger the starting of a recording by setting the do_recording UVar to " start", e.g.:

```
at (dm.event_listenToUser?(var listenToUser)) {
 echo("dm requested we start listening to user with value="+listenToUser);
 vad.do_recording=listenToUser;
};
```

The same UVar (do−recording) can be used for stopping or cancelling a recording by the DM, as described in section C.
A recording starts right away and is terminated when:
a) the speech activity is finished: event_do_recognition is emitted with 3 attributes (the audio filename (including the pathname), a string indicating "voice segment", and a timestamp for logging purposes).
b) there is no speech detected for a period equal to the recorder_timeout_no_speech parameter: event_no_speech_segment is emitted with attribute "time−out" and a timestamp for logging purposes. Note that this event is not emitted when multiple recordings are made (vad.multiple_recording = "true"). In case that several files are produced containing the detected speech segments, only the event_do_recognition is emitted.
c) there is continuous speech activity for a period equal to the recorder_timeout_speech parameter: event_do_recognition is emitted with with 3 attributes (the audio filename (including the pathname), a string indicating "recording_time−out" and a timestamp for logging purposes).

In case the DM request to stop recording before the above mentioned conditions are met:
a) An event event_do_recognition is emitted with 3 attributes: the audio filename (including the pathname), a string indicating "recording_stopped_viaUrbi" and a timestamp for logging purposes.
b) An event event_no_speech_segment is emitted with attribute "stop−requested" and a timestamp for logging purposes.
In case the DM request to cancel recording before the above mentioned conditions are met, no event is emitted and no file is written.

The recorded file to be processed by the ASR is of raw PCM format with sampling frequency equal to the sample frequency of the incoming audio data. The file is placed in the folder defined in vad.u_filepath and the filename has the following format: audio−timestamp−x.raw, where x is a unique recording id number, e.g. audio−timestamp−1.raw, audio−timestamp−2.raw, etc.

The event event_no_speech_segment is addressed to the DM informing that there is no speech input, while event event_do_recognition is addressed to the ASR. The payload (attribute) of each event contains the status of the event (e.g. if a recording has been timed out, the speech utterance might not be complete).
Therefore these event can be linked with the pertinent modules are demonstrated below:

```
// VAD: informing DM that no speech activity was detected
at(vad.event_no_speech_segment?(var reason, var timelog)) {
  echo("vad informs that no speech was recognised because: "+reason);
  dm.setNoSpeechDetected(reason);
};

// VAD: request recognition
at(vad.event_do_recognition?(var filename, var status, var timelog)) {
  echo("vad do recognition event with status: "+status);
  realAsr.doASR(filename);
};
```

****
E. PARAMETERS

VAD Parameters
−−−−−−−−−−−−−−
o Sampling frequency (u_fs): The sampling frequency of the incoming audio stream (Note: it is assumed that a) the signal is quantised at 16bit, and, b) there is only one channel present).

o Window length (u_windowLength): Length (in sec) of the window.

o Window overlap (u_windowOverlap): Fraction of window overlap (e.g. 50% overlap).

o Time for initialisation (u_initLength): Time (in sec) at the beginning of the sound file during which the VAD estimates the initial noise properties. In order for eVAD to work properly, the sound file has to start with a pause. During this time period the input signal should contain no speech.

o Initialisation energy percentage (u_initialSoundPercentage): When the energy characteristics of the first "time for initialisation" (in sec) of the signal are estimated, "initialisation energy percentage" percent of this energy will be taken for the noise energy, so if noise is expected at the beginning, this parameter should be set to e.g. 100. This parameter is introduced in order to cope with sound signal that don t start with a pause.

o Front hangover time (u_frontHangoverTime): When a speech region is detected, also the "front hangover time" seconds before that region will be considered to be speech. Note that this parameter can have some influence on the optimal threshold value, i.e. with a shorter "front hangover time" could be that the threshold has to be lowered.

o Hangover time (u_hangoverTime): When a speech region is detected, also the "hangover time" seconds after that region will be considered to be speech.

o Minimum pause length (u_minPauseLength): Minimal length of a pause. If a pause is detected that is shorter than "minimum pause length" it will be discarded and not displayed in the output, nor used to update any noise properties.

o Minimum speech length (u_minSpeechLength): Minimal length of a speech region. If a speech region is detected that is shorter than "minimum speech length" it will be discarded (considered to be part of the

surrounding pauses).

o Smoothing parameter (u_nrUsedAdjacentFrames): Parameter that indicates how many frames are used to calculate the feature used for the classification.

o Threshold (u_threshold): Value to which the calculated feature will be compared in order to make the speech/ pause decision. If the threshold is variable ("variable threshold" = 1 see below) this will be the starting value.

o Variable threshold (u_varThreshold): If "variable threshold" = 1, a variable threshold that is function of a detected Signal to Noise Ratio (SNR) will be used, if not, a fixed threshold that is equal to the specified threshold value will be used.

o SNR percentage (u_threshSNRpercentage): Percentage of the detected SNR to which the threshold will be set.

o Begin frequency (u_usedFreqBand0): Lower frequency (in Hz) of the frequency band in which the energy is considered

o End frequency (u_usedFreqBand1): Upper frequency (in Hz) of the frequency band in which the energy is considered. If the full spectrum should be used, the "end frequency" parameter should be set to half of the sampling frequency (or bigger) and the "begin frequency" to zero.

o Minimum speech power (u_minSpeechPowerdB): Minimal power (not energy) (in dB) of speech in the given frequency band. If the power of a frame is lower than "minimum speech power" it will always be classified as a pause frame (if the parameters "front hangover time", "hangover time" and "minimum pause length" allow this). By setting the threshold parameter to a very low value (e.g. −100) it is possible to use only this " minimum speech power" parameter to make the speech/pause decision, this is useful in case it is know that background noise power is always lower than a certain value. If nothing is known about the speech power the parameter can be set to a very low value (e.g. −1000) so it has no influence anymore. It is wise however to select a low minimal power that speech is expected to attain in order to make the VAD deaf to very low power sounds. This is especially important if the VAD has to be able to cope with sudden noise loudness changes.

o Noise estimation smoothing parameter (u_alphan): Parameter that determines how the noise properties are updated. The higher this value the more weight is given to the previous estimation. Its value must be in the region [0,1]. This parameter can have an influence on the optimal threshold value.

o SNR estimation smoothing parameter (u_alphap): Parameter that determines how the detected SNR is updated. The higher this value the more weight is given to the previous estimation. Its value must be in the region [0,1].

Recording Parameters
−−−−−−−−−−−−−−−−−−−−

o Target directory (u_filepath): it is where the audio files are written.

o Speech timeout period (recorder_timeout_speech): it is the time (in sec) after which the recording will stop (in case there is continuous, nonstop speech).

o Silence timeout period (recorder_timeout_no_speech): it is the time (in sec) after which the recording will stop (in case there is no speech activity present).

o Minimum length of audio (u_minAudioPortion): it is the minimum audio length (in sec) that can be recorded in a file. Audio segment of shorter length are ignored.

o Recording of multiple audio segments (multiple_recording): When this parameter is set to "true", the VAD will create multiple files, each one containing the detected speech segments. See section C for more info.

7.12 Voice Modification README

```
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
| ALIZ−E Project − Voice Modification for NAO |
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
Author: Giorgos Athanasopoulos
Email: gathanas@etro.vub.ac.be
Last update: 02.12.2013

BASIC REQUIREMENTS:
g++ 4.4.x
URBI SDK 2.7.5
Note: the Voice Modification has been tested in Ubuntu 10.10 & 12.10, 32bit.

*************
A. PREREQUISITES

    This module makes use of the existing Audio Front End (AFE) module. Make sure you have the AFE
   source code available.
    For the gstreamer (used both by the AFE and for streaming the modified voice to NAO), the following
   packages need to be installed (normally they should be already installed during the standard Alize scenario'
   s installation):

    $ sudo apt−get install libgstreamer0.10−dev libgstreamer−plugins−base0.10−dev gstreamer0.10−x
   gstreamer0.10−plugins−base gstreamer0.10−plugins−good gstreamer0.10−plugins−ugly−multiverse
   gstreamer−tools

    In addition, the following Voice Modification specific packages need to be installed:

    $ sudo apt−get install libsamplerate0 libsamplerate0−dev

*************
B. COMPILING:

    The following environment variables need to be set (e.g. in $HOME/.bashrc or $HOME/.profile):
    export URBI_ROOT=/YourPathHere/urbi−sdk−2.7.1−linux−x86−gcc4
    export PATH=$URBI_ROOT/bin:$PATH

    NOTE: It appears that the modules is not working when compiled with the Urbi version provided by Remi
    (i.e., urbi−sdk−2.7.5−boost1.45−release). Instead, the module must be compiled and ran using the
   standard Urbi version (you can you download it from http://www.gostai.com/downloads/urbi/2.7.5/ ).

    Copy (e.g. from svn) the source files in e.g. ~/YOUR_VoiceModification_PATH directory. In the same
   directory copy the source files of AFE module (i.e. GstAudioFrontEnd.u, UGstAudioFrontEnd.cpp,
   UGstAudioFrontEnd.h).

    You can now compile all components in a single .so object by:
    $ cd ~/YOUR_VoiceModification_PATH
    $ g++ −Wall $(pkg−config −−cflags gstreamer−0.10 samplerate) −I $URBI_ROOT/include −fPIC −
   shared *.cpp −o NAOvoice.so $(pkg−config −−libs gstreamer−0.10 samplerate) −lgstapp−0.10

    The NAOvoice.so file should be now created in your ./YOUR_VoiceModification_PATH.

*************
C. CONFIGURATION:

    Before running the Voice Modification module please follow the following steps:

    1. If NAO is to be used, you need to enable RTP/UDP streaming on the robot by installing the libgstudp.
   so file. Please file the instruction from here: http://alize.gostai.com/wiki/wp4/
   How_to_enableuse_Gstreamer_and_RTPUDP_plugins_on_Nao

    2. Modify GstAudioFrontEnd.u for setting the appropriate output buffer size in bytes: var OutBufferSize =
    1600; (this is already done in the provided file)

    3. In VoiceModificationNAO.sh set the Urbi server IP addrress (e.g. NAO's IP address) and port (e.g.
   54000).
```

4. In VoiceModificationNAO.u set the the IP−address where the receiving UDP/RTP server is running (e.g . NAO's IP address) and the listening port (e.g. 5002).

5. In VoiceModificationNAO.u you can also set the Voice_Modification_Factor parameter. A value of 1 should leave the WoZ voice unmodified. Values smaller than 1 (e.g. 0.7) will result in a higher pitch voice, while values greater than 1 (e.g. 1.2) will result in a lower pitch voice. The parameter accepts values in the range (0 − 3), otherwise the value is set to 1.

6. In order to increase the intelligibility of the modified speech (which is affected by the quality of Nao's loudspeakers), post−processing equalization is applied. The core idea is to increase the gain for high frequencies and reduce it for lower ones (where loudspeaker distortions are more prominent). For this, a gstreamer 10−band equalizer is used. The gain for each band can be set in VoiceModificationNAO.u and the expected value is in the range [−24,12] (that is in dB). If you do not wish to use equalization, you may set all gains to 0.

Note: if the parameters of steps 5 & 6 are modified while the module is running, the new values will be taken into account only after the voice modification module has been restarted.

****
D. RUNNING THE MODULE:

1. Set up an Urbi server:
$ rlwrap urbi −H localhost −P 54000 −i
(on NAO you can run naoqi with urbi enabled, please refer to Aldebaran SDK for more information).

2. From a different shell you need to start the listening GSTREAMER UDP/RTP server.
For localhost:
$ gst−launch−0.10 −v udpsrc port=5002 ! "application/x−rtp,media=(string)audio,clock−rate=(int) 16000,width=16,height=16,encoding−name=(string)L16,channels=(int)1,payload=(int)96" ! rtpL16depay ! audioconvert ! alsasink sync=false
On NAO (you should connect using ssh):
$ GST_PLUGIN_PATH=/home/nao/gst−plugins∼lenny2_i386/ gst−launch−0.10 −v udpsrc port=5002 ! "application/x−rtp,media=(string)audio,clock−rate=(int)16000,width=16,height=16,encoding−name=(string)L16,encoding−params=(string)1,channels=(int)1,channel−positions=(int)1,payload=(int)96" ! rtpL16depay ! audioconvert ! alsasink

Note: the GSTREAMER UDP/RTP server port (port=5002 in the examples above) should be the same as the one set during step 4 of section C. CONFIGURATION.

3. From a different shell launch the Voice Modification module:
$ cd ∼/YOUR_VoiceModification_PATH
$ ./VoiceModificationNAO.sh

4. If no errors were reported, the module should be running. Talk to the microphone and hear your modified voice through NAO's speakers!

****** END ******