SEVENTH FRAMEWORK PROGRAMME

SIGNSPEAK

SignSpeak Project

Scientific understanding and vision-based technological development for continuous sign language recognition and translation

SignSpeak technology demonstrator

Minor Deliverable D.6.2

Release version: V 1.0

Grant Agreement Number 231424

Small or medium-scale focused research Project (STREP)

FP7-ICT-2007-3. Cognitive Systems, Interaction, Robotics

Project start date: 1 April 2009

Project duration: 36 months

Dissemination Level		
PU	Public (can be made available outside of SignSpeak Consortium without restrictions)	
RE	Restricted to SignSpeak Programme participants and a specified group outside of SignSpeak consortium	X
IN	SignSpeak Limited (only available to a specified subset of SignSpeak programme participants)	
LI		
Distribution list (only for RE or LI documents)		

0 General Information

0.1 Document

Title	SignSpeak technology demonstrator
pe	Minor Deliverable
Ref	D.6.2
Target version	V1.0
Current issue	V1.0
Status	Draft
File	D_6_2_v1.odt
Author(s)	Pablo Alonso-Villaverde Roza / CRIC
Reviewer(s)	Gregorio Martínez / CRIC
Approver(s)	Gregorio Martínez / CRIC
Approval date	
Release date	25/05/12

0.2 History

Date	Version	Comment
25/05/12	V1.0	Released first description of D.6.2 report

Authors	Group
Pablo Alonso-Villaverde Roza	CRIC

0.3 Document scope and structure

One of the tasks that WP6 addresses, is the integration of the software development made in WP3, WP4 and WP5 for SignSpeak.

That integration will be made by means of another software module, an integration framework. This framework will be also used as demonstrator of the technology and work done in SignSpeak.

This document (D.6.2) covers technical aspects of the work done in that integration.

There are other reports (D.6.1 and D.6.3) related to the general framework design and the Graphical User Interface for non-technical users respectively.

Table of Contents

1 Overview

This document deals with the framework developed for the integration of the different work packages of SignSpeak. The integration framework will serve both as a software integration tool and as a technology demonstrator for SignSpeak.

There is a brief introductory video about the integration framework uploaded in YouTube http://www.youtube.com/watch?v=hbDUcJKzmJ0

Design considerations for this integration framework have been explained in the report D.6.1. of SignSpeak. This document focuses on the technical approaches taken to implement the most important parts of the integration framework.

2 Integration framework

The main objective of the integration framework is connecting the software modules developed in different work packages of the project, so they can work as a whole.

It can also connect software modules developed by different researchers and institutions, so all the modules can work as a whole in different configurations or pipelines, so researchers can see how that affects to the results of their own modules.

The next illustration shows a full translation pipeline of SignSpeak. It shows the main components of the project connected in a graph-like structure.

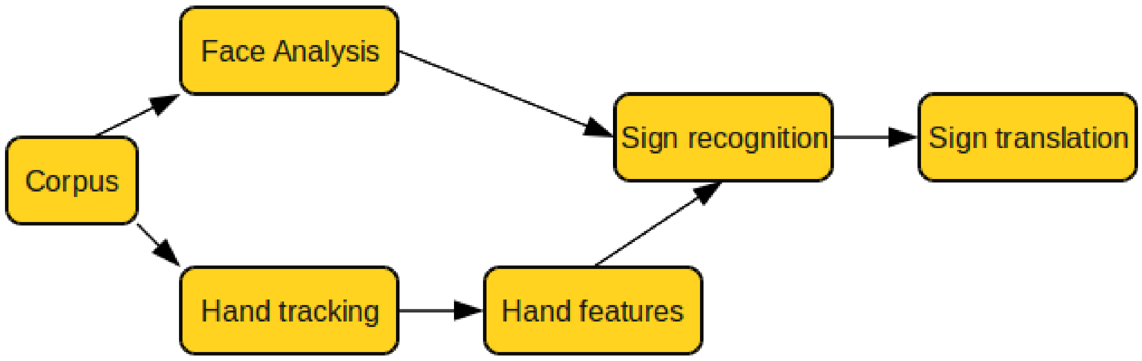


Illustration 1: Full translation pipeline

The framework is not limited to work with a full system. It can be used to run or test parts of the system. This is useful when debugging or fine tuning a specific software module. For instance, we could run the hand tracking software with a diagram like the following.

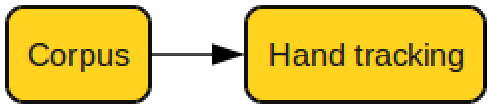


Illustration 2: Tracking subsystem

The framework allows setting as arguments to software modules, data from other modules or alternatively data existing on disk. If we had data (generated by ourselves or given to us by other means), we could run, for instance, sign recognition and translation software without running other modules usually present in the full translation pipeline, like tracking or feature extraction.

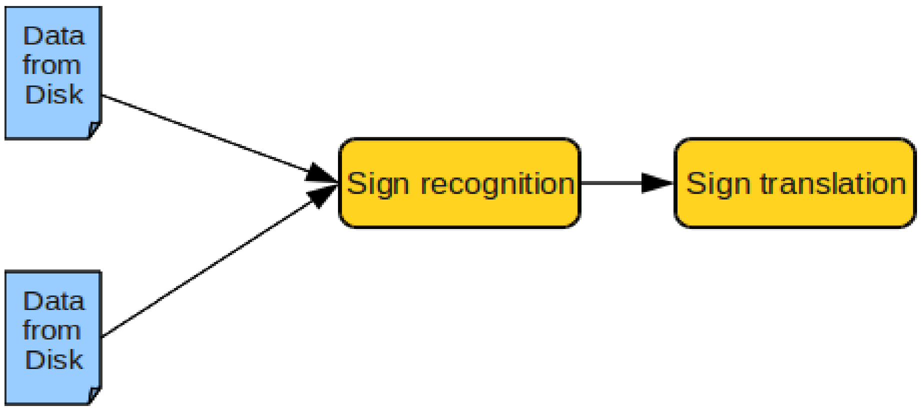


Illustration 3: Running sign recognition and translation

The framework is not limited to running the modules initially designed for SignSpeak. New software modules can be added to the framework to be run if desired. These new modules can be used to improve or add new features to SignSpeak. For instance, we could add a new module to translate from German to English language.

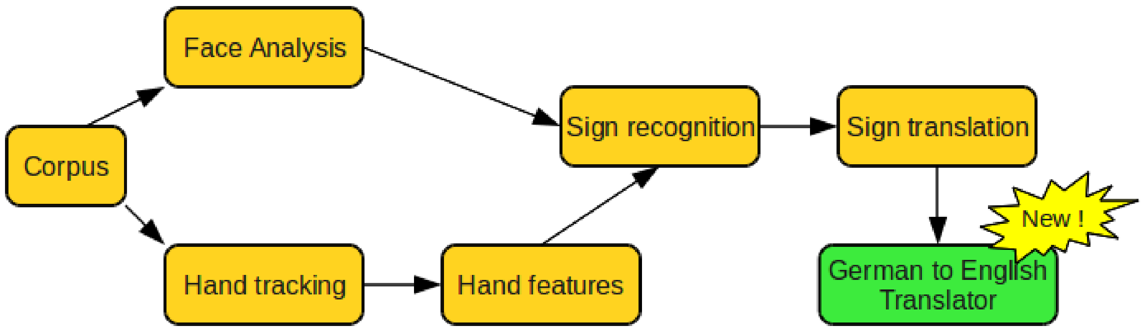


Illustration 4: Adding a new module

Later in the document, it is explained how to add new software modules to the integration network.

Unlike other work packages (WP3, WP4, WP5) in SignSpeak, this is not a research package, it is a pure development package.

2.1 General overview of the integration framework

As described in the previous report D.6.1, the integration framework will be another software module, that will allow the connection of other software modules visually by means of a graph or network diagram.

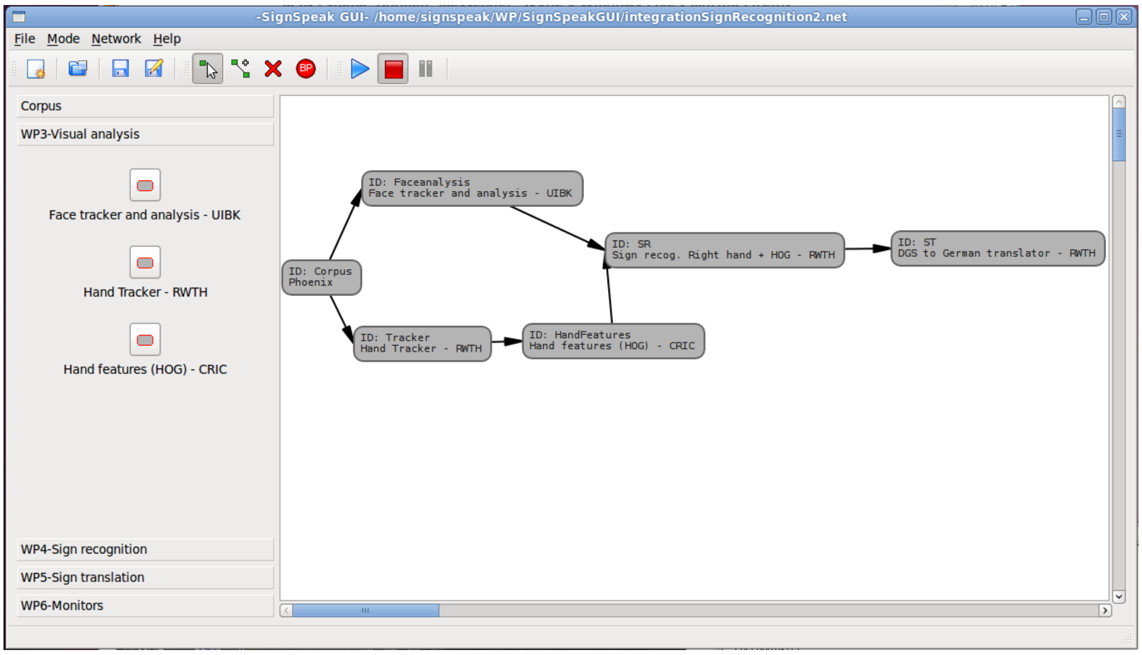

-SignSpeak GUI- /home/signspeak/WP/SignSpeakGUI/integrationSignRecognition2.net
File Mode Network Help
Corpus
WP3-Visual analysis
Face tracker and analysis - UIBK
Hand Tracker - RWTH
Hand features (HOG) - CRIC
WP4-Sign recognition
WP5-Sign translation
WP6-Monitors
ID: Faceanalysis Face tracker and analysis - UIBK
ID: Corpus Phoenix
ID: SR Sign recog. Right hand + HOG - RWTH
ID: ST DGS to German translator - RWTH
ID: Tracker Hand Tracker - RWTH
ID: HandFeatures Hand features (HOG) - CRIC

Illustration 5: Integration framework

Using a graph structure has several advantages:

- Fast overview of the full system
- Easy ordering of modules
- Ability to check for design mistakes
- Control over the execution of the full system
- Flexible design and modification of the system to be represented
- Easy to expand.

For further details on these advantages, please refer to the report D.6.1.

Each node in the graph will represent a software module and a connection between two software modules (nodes of the graph), will represent a dependency. That dependency means that the software module at the end of the connection needs data generated by the software module at the beginning of the connection.

After the user of the framework has designed the graph, he or she will be able to change execution arguments for each individual software module. The user will be able to set bindings among the information generated by a module, and the information needed by another module can be connected to the previous one.

Once the graph is designed and execution arguments set, it is possible to run an execution of the full graph according the order defined by the structure of the graph.

The user can check the results of each software module in the locations specified in the arguments of each module or use the monitors and visual tools included with some modules.

This software requires technical skills and some knowledge about the automated translation process. Although the software has been simplified, it is not aimed at non-technical users.

In the following sections of this report, the main technical aspects related to the implementation of this software will be explained.

2.2 Development of the integration framework

In this section, we will enumerate the main parts of the integration framework, paying special attention to the tools and the methods used for their implementation.

2.2.1 General development tools

The integration framework has been developed on Ubuntu Linux 10.04 - 32 bits. Linux was a common operating system for some partners and was also the initial development platform for some software modules of the project.

The initial prototype of the framework, was developed with the Python interpreter included by default in this operating system (Python 2.6.5). Python is an interpreted programming language, widely used in Computer Science. It lacks the speed of compiled languages like C or C++ but the development is quite easy and fast. There are lots of libraries and tools available and its performance is good enough to develop this kind of software.

This language proved to be a good option when combined with Qt 4.6.2 (a multiplatform library used to develop user interfaces). As such, it was decided to keep using Python and Qt. To connect Python and Qt we have used PyQt, a library that allows programs written in Python to use the Qt library.

The editor used to program was Geany, although there are many other possible options. In fact, is quite easy to use other editors if desired. The development is not tied to any other tool besides Python, PyQt and Qt.

2.2.2 User Interface development

The interface of the framework is made of a single window with standard menus and a buttons bar at the top of the window, a palette on the left side of the window and a drawing zone of the right side of the window.

The main window was created with the Qt class QMainWindow.

2.2.2.1 Menus and buttons bar

Usual operations like opening or saving a file, or the operations to work with the nodes of the graph are located in the menus and duplicated in the buttons bar.

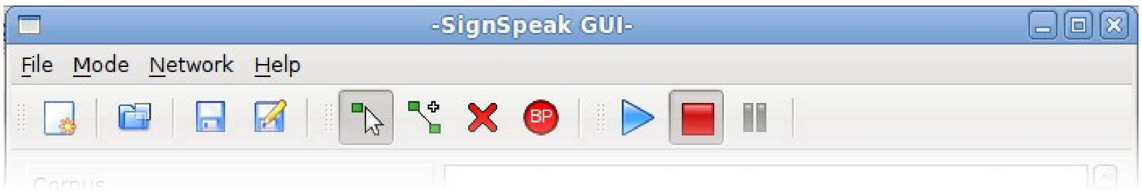


Illustration 6: Menus and buttons bar

Menus and button bars were created with the usual Qt components for these user interface elements like QMenu, QBar. The actions carried out by these elements are set by means of QAction objects.

2.2.2.2 Palette of modules

On the left of the window, there is a palette divided into several sections. Each section contains several buttons that represent software modules. The buttons of this palette are used to create the combination of nodes (software modules) that make up the graph.

Each section of this palette matches a work package, so we can find easily the software module we want to add to the graph.

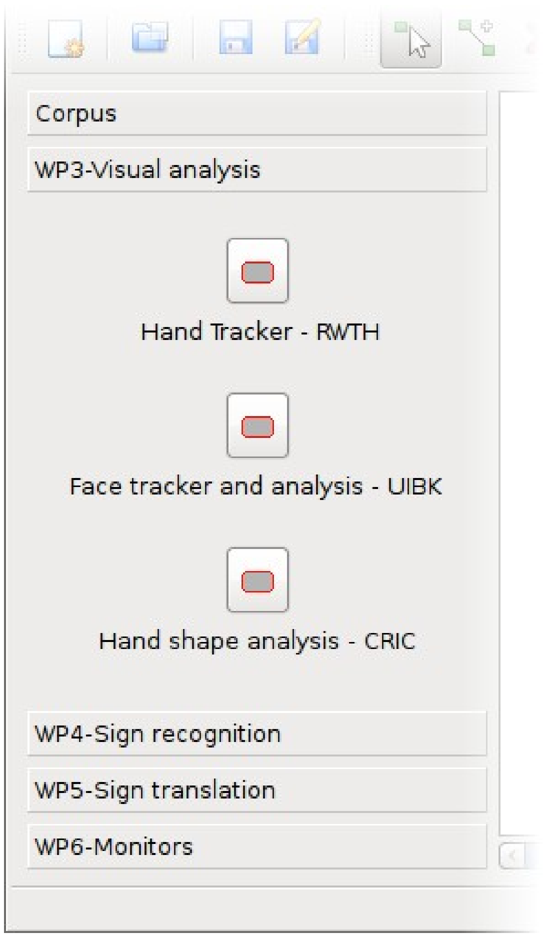


Illustration 7: Palette

The palette was created using a QToolBox component. The groups were created with QButtonGroup components and the buttons inside each group with QToolButton components.

2.2.2.3 Drawing zone

On the right of the main window, there is a drawing zone, where the user can design the graph with the elements available in the palette and the operations available in the menus and button bars.

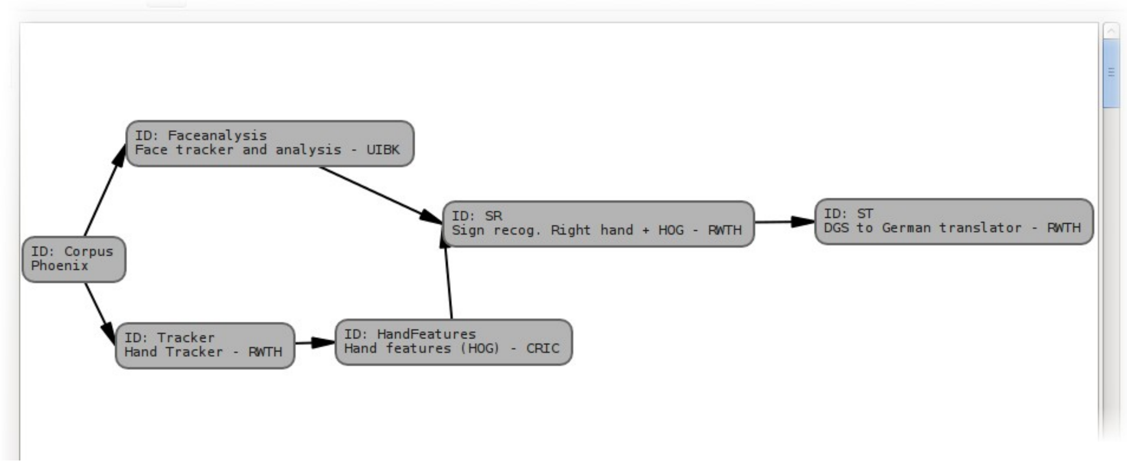

ID: Faceanalysis
Face tracker and analysis - UIBK
ID: SR
Sign recog. Right hand + HOG - RWTH
ID: ST
DGS to German translator - RWTH
ID: Corpus
Phoenix
ID: Tracker
Hand Tracker - RWTH
ID: HandFeatures
Hand features (HOG) - CRIC

Illustration 8: Drawing zone

The drawing zone was created using different Qt components. A QGraphicsScene that has functions to render different shapes and lines, a QGraphicsView that offers a screen surface to draw the shapes created with the QGraphicsScene component.

The QGraphicsScene component can also detect the mouse actions of the user (moving the mouse, clicking on an element of the screen, etc.) so it is also used to track user actions carried out when drawing the graph.

2.2.3 Implementation of the graph

The graph of modules was implemented using a typical graph data structure based on adjacency lists. The graph component stores:

- A list of the nodes that make up the graph.
- An adjacency list, that relates a node with the nodes it connects.
- A list of connection objects, that represent the connections between nodes.

The graph operates on nodes. These nodes were implemented as generic data containers, so it is possible to manage any kind of information in the graph. The graph does not have access to the data being managed. It only focuses on keeping nodes correctly organized as a graph.

Like most graph structure implementations, it offers the usual operations needed by graphs like adding or deleting nodes, checking if a connection between two nodes exists, etc.

A node, as mentioned before, is only a generic data container, so a node could store any kind of information. Nodes store an internal value used as an identifier, which can be used to quickly identify the node from inside or outside the graph structure.

The real data relative to a module is stored in a NodeData object. This object stores all the necessary data to represent a software module (name of the executable, arguments, and their data types).

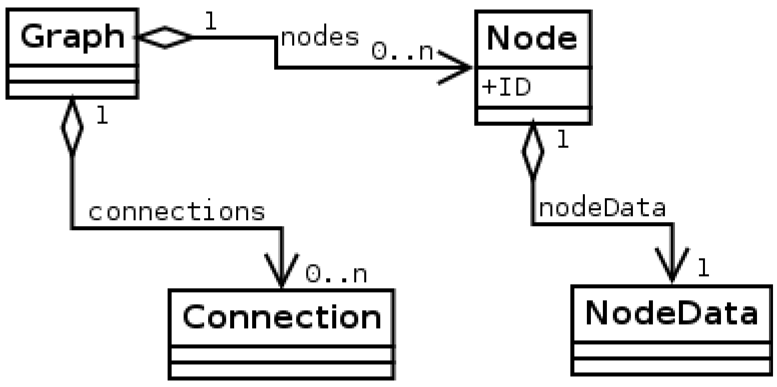


Illustration 9: Graph and Node components

Some of the components described above, specifically the Node and Connection also need a visual representation on the screen. That visual representation has been made by means of of a QGraphicsItemGroup component. This component from Qt, allows drawing a set of shapes, texts and lines on the screen, so it was a perfect for that need. In the framework code, we have derived classes from QGraphicsItemGroup to make visual representations of nodes and connections, respectively and whenever a new node of the graph is created, a graphical node is created too, and both are related by means of a dictionary that uses the node identifier as an index to locate its graphical representation.

Connections use a similar system, there is a match between a connection object in the graph and the object that makes the visual representation of the connection on the screen.

2.2.4 Representing software modules

Using different software modules with variable features, generates some problems that must be solved. One of these problems is the specific needs of each software module relative to the user interface. Another problem is how to make a proper call to each module taking into account the different nature of executables. In the next sections we will explain how these two problems were solved.

2.2.4.1 User interface modifications for each module

The integration framework must be able to manage software modules from different work packages. First of all, these software modules have different number and type of arguments. For example these arguments could be:

- File names.
- Folder names.
- Numeric values (integer or floating point values).
- Boolean values.
- Etc.

As such, a single interface to set arguments to every software module is not valid. It must be changed dynamically depending on the module.

In order to carry out the previous operation the framework stores a definition for each software module in a XML file named “NodeConfiguration.xml”. In this file, there is information about the module name, number and type of parameters, their default values and descriptions of each argument to help the user.

When the framework starts, it loads that file and reads the information about the executables, so it's possible to take the executables' details into account inside the framework.

If we want to add more modules to the framework, we can add new entries to that file and the new modules will be shown on the palette of the framework to be used as nodes of a graph.

That definition helps us to configure automatically the user interface according to the module we are working on. See the next illustration.

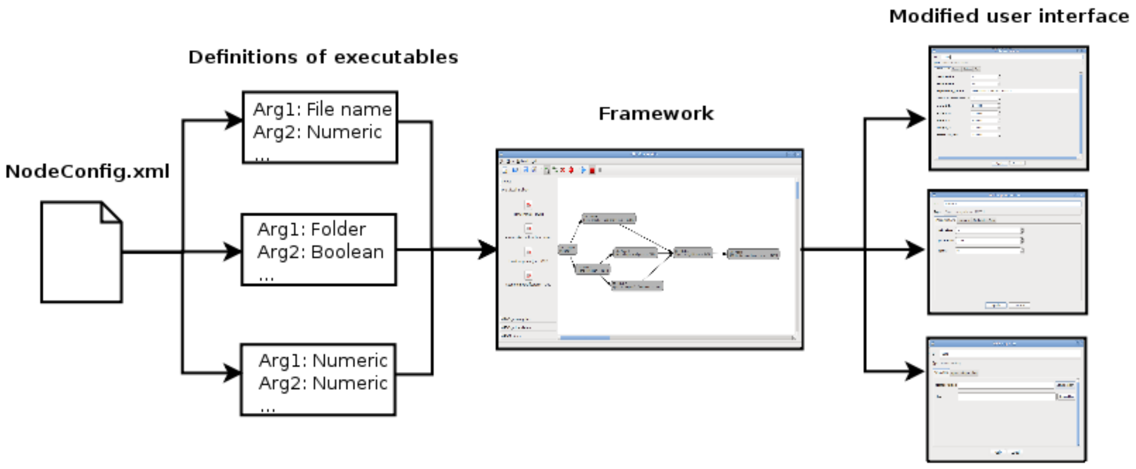
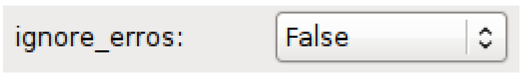


Illustration 10: User interface is modified depending on the module and its arguments

When a user wants to set arguments to one of the nodes (software modules) the framework creates an “empty” dialog window that is dynamically populated with interface elements specific to each argument of that module. For example, if one of the arguments is a boolean value, the framework will show a combo box next to the argument name that allows the user to choose between “true” and “false”. If the argument is a numeric value, it will show a text box modified to enter numeric values, and so on.

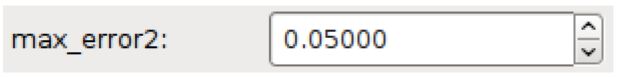
The types of arguments currently supported by the framework are:

- **Boolean**. Can take values “true” or “false”. We can set the value of a the argument by means of a combo box. For instance, this is an argument to ignore error messages in a module:



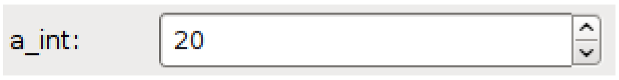
ignore_erros: False

- **Numeric (floating point)**. A floating point numeric value (positive or negative).



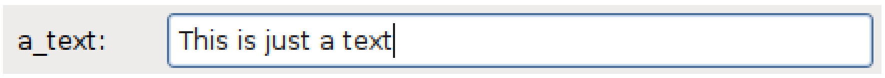
max_error2: 0.05000

- **Numeric (integer)**. An integer value (positive or negative).

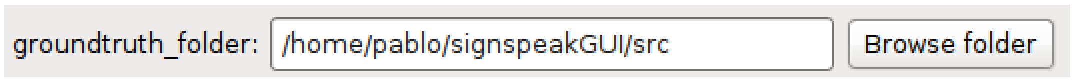


a_int: 20

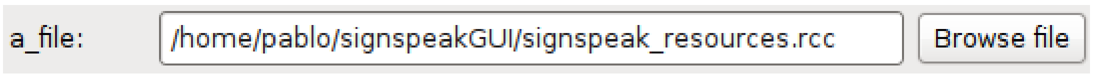
- **Text or string**. A simple string. A standard text box is used to set its value.



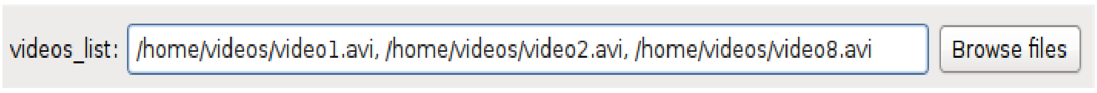
- **Folder or directory**. These argyments can be set by means of a browse folder dialog (pressing the button on the right) or by writing the path directly in the text box.



- **A file**. It can be selected with a file browser dialog pressing on the button on the right or by writing directly in the text box.

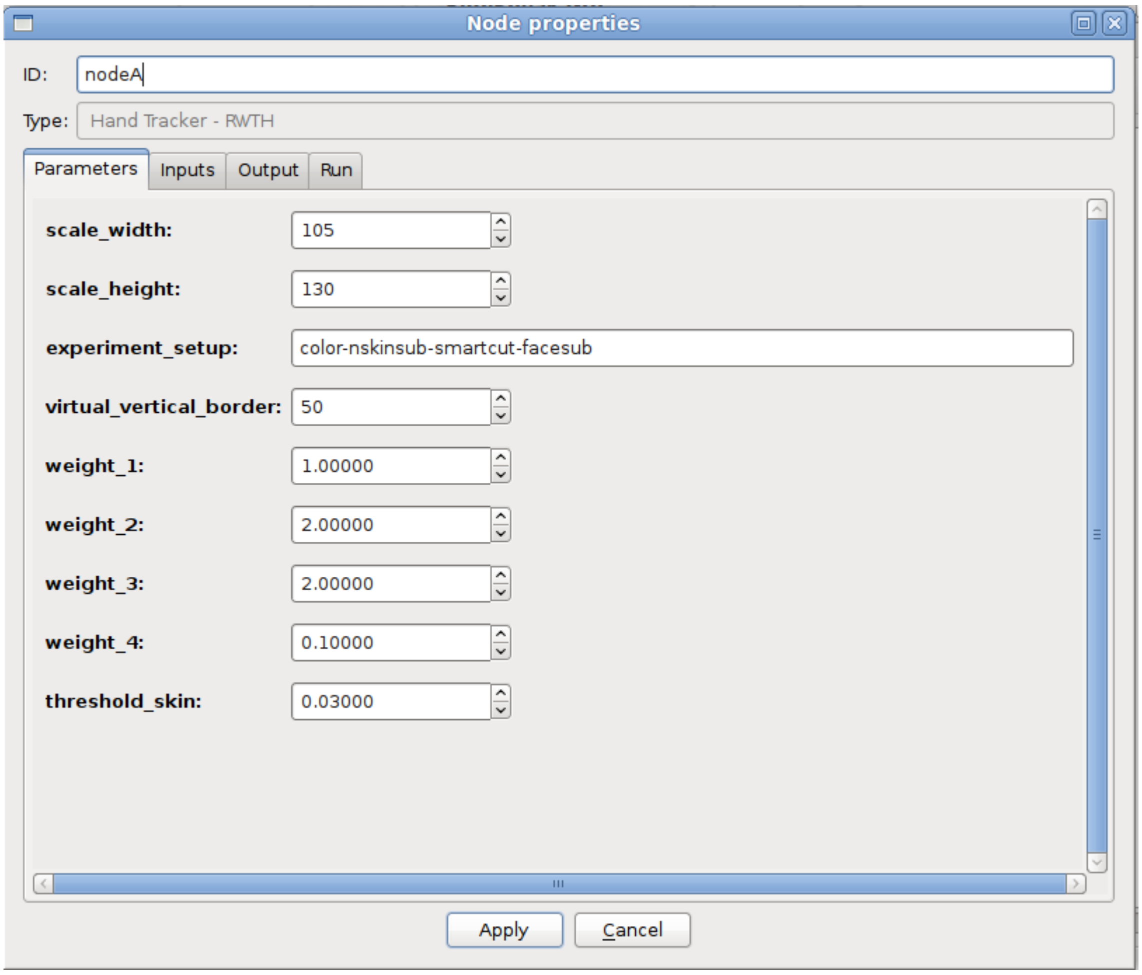


- **List of files**. A variable list of file names, separated by comma. They can be selected by a browser or written directly in a text box.


videos_list:
/home/videos/video1.avi, /home/videos/video2.avi, /home/videos/video8.avi
Browse files

- **Parameters group.** This kind of argument, can take as values different sets or groups of variable parameters. The program allows selecting one of those sets and editing its parameters. For example an arguments to select sampling options for image processing could be chosen among different sampling methods (Corner, Hessian, Star, Random, Contour) and each method could have its own parameters with specific values.

In the next image we can see a combination of arguments in a window generated dynamically by the framework to match the arguments needed by a specific software module.



ID: nodeA
Type: Hand Tracker - RWTH

scale_width:	105
scale_height:	130
experiment_setup:	color-nskinsub-smartcut-facesub
virtual_vertical_border:	50
weight_1:	1.00000
weight_2:	2.00000
weight_3:	2.00000
weight_4:	0.10000
threshold_skin:	0.03000

Illustration 11: User interface to set arguments of a tracker module

2.2.4.2 Executing a software module

Even if we have the right arguments defined, the framework also needs to use these arguments with the right syntax, so the modules can be called and run.

The nature of executables of each module is quite different. There are:

- Conventional executables
- Shell-scripts
- Matlab programs

To solve the syntax problem when calling an executable the framework will run each executable by means of an intermediate shell-script. That shell-script will adapt the arguments given by the framework to the right syntax and will run the executable.

Using these shell-scripts have some advantages. For instance, it allows running additional processes before and after the main executable of the software module. This feature could be useful to carry out transformations on the information generated by the executable like translating file formats, change working directories, etc.

Another advantage, is setting callback routines that will locate all the processes generated by the software module and stopping them properly when the framework needs to force a stop on the execution of the software module.

There is a subfolder named 'launchers' in the directory of the framework that contains shell-scripts to launch each software module. If we add a new software module to the framework we can create a new shell-script using a generic example given as a template in that folder or we can also use one of the existing ones, if we prefer. After that, we must refer that script in the module definitions file (NodeConfig.xml) and the framework will call the module automatically if we use it in one of the graphs we design.

2.2.5 Execution of the graph

When the user has created a graph or network of modules it can request the framework to run it. Running a graph involves:

- Generating an ordering of the modules (nodes) in the graph.
- Running each module (node) according to the ordering and using the arguments set to the module.

2.2.5.1 Generate an ordering

In order to run the software modules in the graph we need to create a linear ordering of the modules (nodes) in the graph. That ordering depends on the structure of the graph (how the nodes have been connected).

This ordering is necessary because some modules need data generated by other modules. We cannot run the modules in just any order. We must define the order of execution based on the connections of the graph. Sometimes there is more than one valid ordering for the same graph, that's not a problem.

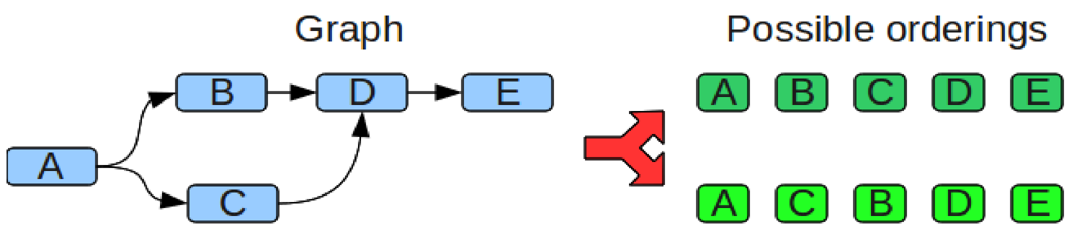


Illustration 12: Orderings of a graph

To get a correct linear ordering of the nodes of the graph, there are graph-specific algorithms that can be applied to our graph. The algorithm used to deduce the ordering is a “Topological ordering traversal”.

This algorithm will also detect some common mistakes like cyclic dependencies. Taking a look at the graph in the next illustration, we can see a cyclic dependency to be avoided, because three modules (A, B and D) are interdependent. Interdependency is not allowed, because it is not possible to get an execution order. It is not possible to know which module must be run first. In these cases we ask the user to remove the cycle so there is no ambiguity about the execution ordering.

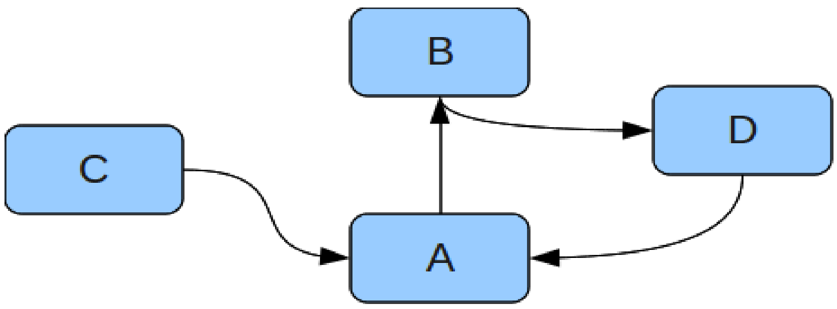


Illustration 13: Graph with a cycle (A,B,D)

2.2.5.2 *Running the ordering*

Once the framework has calculated a linear ordering of execution for the graph, it starts a secondary thread that will run the ordering. The thread is necessary to avoid blocking the user interface of the framework with the intensive calculations performed by some modules.

The secondary thread will receive the ordering from the main process and will iterate it until it reaches the last module or a until stop order is received from the user interface.

For each module to be run, the thread extracts the argument values for that module and calls the launcher shell-script described in the section 2.2.4.2

Each shell-script is run by means of 'Popen', a Python function from the 'subprocess' module of Python. This methods allows the framework detect when a module has finished and continue with the next one in the ordering.

2.2.5.3 *Connecting the execution of modules. Data flow.*

In most cases, when a software module of the graph is executed, it depends on existing information previously generated by another module, but it is not only a matter of having the information, the location of that information is necessary too or the following modules will not be able to run properly.

In the framework we have implemented a simple system to make the locations of information flow along the graph.

In order to achieve this, a module must classify each argument into one of the following types (Input, Output or Common parameters).

- Inputs: These arguments refer to information that must exist before running the module, and in most cases it has been previously created by another software module in a file, a set of files or a folder.
- Outputs: These arguments specify where a module must store the information it generates. It is also a location on the disk.
- Common parameters: These are the rest or arguments or parameters of the program, necessary to run the module but they don't depend on other modules. For instance a value to enable or disable a specific algorithm or values for different settings of the module.

This classification is made once, in the definition of the module.

Later, after finishing the design of a graph and setting arguments to the modules, those arguments classified as “inputs” can be binded to “output” arguments of previous modules in the graph, so a module can automatically know where to look for the needed information. The framework, will move values from outputs of modules to the inputs of other modules binded to those outputs.

When setting the arguments of a module, the user can choose between setting values of an argument manually or taking the values from a previous connected module, as long as the types or arguments are compatible.

2.2.6 Saving and loading graphs

The graphs designed in the framework can be saved for future use in a file that can be loaded later. The system to save or load our work is not different from most computer programs nowadays. There is a “File” menu that allows the user to save the current graph or load an existing one from a file.

It is important to keep in mind that if, for instance, we modify the file that contains the module definitions (See section 2.2.4.1) , change the number or type of arguments of an existing module, or delete a module definition, the saved files could load partially or generate errors because the framework would not know how to fit the loaded information into the new module definitions. On the other hand, we can add new module definitions without problems. Our saved files will be able to load without problems.

These files are written in XML with the lxml library for Python, and they basically contain the information needed to reconstruct the graph, i.e.:

- A list of the nodes of the graph.
- Values of the arguments set for each module (node) of the graph.
- List of connections between nodes of the graph

2.2.7 Monitors

Monitors are programs used to help in the development of the software modules of SignSpeak. They are used to show information, verify results and check for errors from modules.

Some software modules, like the module for face analysis, include their own monitoring system, others like the hand tracker don't.

For the hand tracker system we have developed a monitor that draws the tracking information on top of the processed videos. That can be used to verify if the tracking process is done correctly. If we play a video in the monitor with tracking information loaded will see how the tracker follows the dominant hand of the signer.

In the monitor, we can play the video at different speeds, pause, advance frame by frame, and move forward and backward to be able to check the results in detail.

This monitor was developed in C++ using Qt 4.6.2 to develop the user interface using QtCreator as development environment.

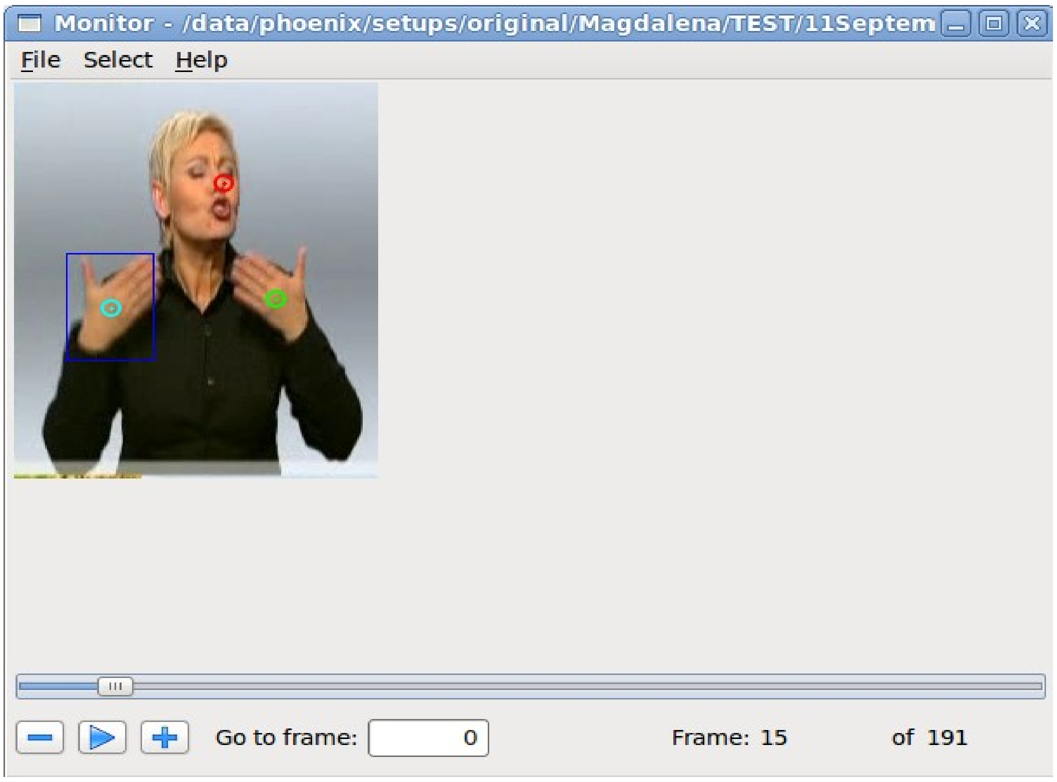
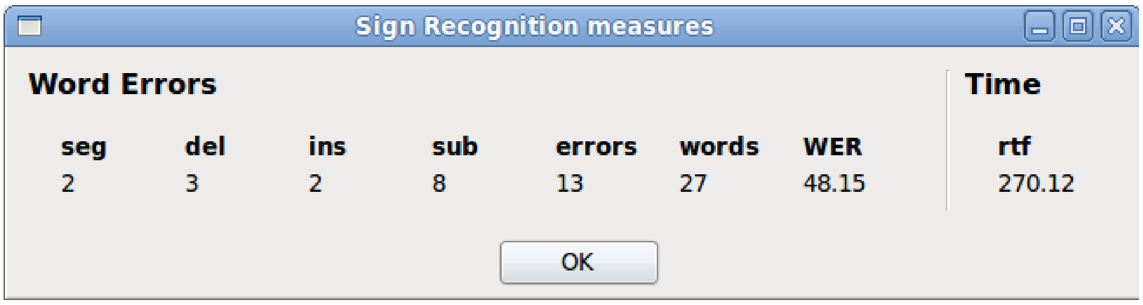

Monitor - /data/phoenix/setups/original/Magdalena/TEST/11Septem
File
Select
Help
Go to frame:
0
Frame: 15
of 191

Illustration 14: Monitor for hand tracking

For other stages of the pipeline, like sign recognition, a monitor would not offer much help, so instead of a monitoring tool, the framework offers a utility to improve the visualization of measurements.

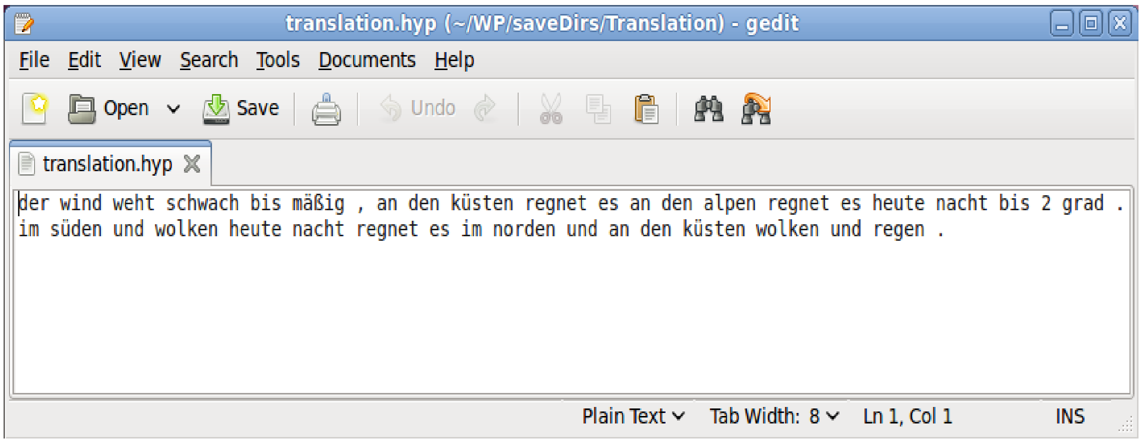


Sign Recognition measures

Word Errors							Time
seg	del	ins	sub	errors	words	WER	rtf
2	3	2	8	13	27	48.15	270.12

Illustration 15: Some measures from the sign recognition process

In the case of sign translation, the final translations of the videos are shown in a text editor.


translation.hyp (~/WP/saveDirs/Translation) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
translation.hyp
Plain Text Tab Width: 8 Ln 1, Col 1 INS

```
der wind weht schwach bis mäßig , an den küsten regnet es an den alpen regnet es heute nacht bis 2 grad .
im süden und wolken heute nacht regnet es im norden und an den küsten wolken und regen .
```

Illustration 16: Final translation of two videos

3 Conclusions

In this report, it is explained how the integration framework contributes with its features to test and run full or partial translation pipelines for SignSpeak. It simplifies the inner configuration of most deliverables, allowing an easier connection between them. Additionally, it allows potential improvements due to the capability of adding new software modules when desired.