Alma Mater Studiorum
University of Bologna

II School of Engineering of Cesena

A Jason & CartAgo MAS to automatically understand academic papers' content: ReadIT4mE!

Project & Development by:

Stefano Mariani (stefano.mariani3@studio.unibo.it)
Mattia Occhiuto (mattia.occhiuto@studio.unibo.it)

Indice

Reasons & Objectives

Sometimes, the best ideas arise from the worst situations...or at least the developers of ReadIT4mE! hope so, because the idea for the project was born in such a bad mood :)

We were working on our thesis, reading tons of useless papers just because the title was not very self-explanatory so we could not figure out what such papers will talk about...but suddenly we said to each other: Why do we have to read all these useless stuff!? What if someone could do this in place of us? What if someTHING instead of someONE??.

Then ReadIT4mE! came to life :)

Apart from jokes, we aimed at creating a system capable of automatically extract key-phrases from academic papers in a simple and fast way, taking a small amount of pre-existing examples to train and learn a proper model for subsequent text mining. Our main objectives were ease of use for non-technical people, involve the end-user asking for feedbacks on text mining engine work and maximize performance. Accuracy of extracted key-phrases was too in the list, but it depends more on the quality of the tool used for extraction than on our usage of it.

The remainder of the paper is structured as follows:

- in chapter 1 we give a brief overview of the system, describing the main entities living in it as well as the tools and frameworks used;
- in chapter 2 we list all the agents living in the system describing their behaviour with a little help from the code;

- in chapter 3 we do the same for the artifacts used by such agents;
- in chapter 4 we highlight the main flows of communication among agents and artifacts in the system, trying to exhaustively explain how the system works;
- then we conclude with some talk about future works.

Capitolo 1

System overview

In this chapter we list existing frameworks and tools used to build the ReadIT4mE! system: this will be done not to explain the concepts behind such technologies nor to introduce them in all their features, but rather to share wiht the reader the motivations behind our choice to use them. Moreover, we highlight all the different agents and artifacts involved in the working cycle of ReadIT4mE! and how they are related to each other.

1 Tools & Frameworks

Let's start with Jason: it is a very comprehensive agent platform embedding an own programming language wich is an extension and a practical implementation of AgentSpeak(L) and a BDI engine to execute plans and manage beliefs and messages. These two features of Jason are exactly the reasons that prompted us to choose such platform: the language is very intuitive and simple and more Prolog-based, while the BDI engine has a reasoning cycle that ease concurrent co-existence of agents in the same environment.

Let's see such reasoning cycle so to better understand developers motivations:

1. Perceiving the Environment;
2. Updating the Belief Base;
3. Receiving Communication from Other Agents;
4. Selecting Socially Acceptable Messages;

5. Selecting an Event;
6. Retrieving all Relevant Plans;
7. Determining the Applicable Plans;
8. Selecting one Applicable Plan;
9. Selecting an Intention for Further Execution;
10. Executing one step of an Intention.

The crucial aspect that more than the others has attracted us toward Jason is that such reasoning cysle is looped over and over without any blocking point: this means that even while executing a certain plan a Jason agent remains responsive to beliefs perception, and more it can start executing other plans concurrently too!

Of course, there are technical issues too that drived us toward Jason platform: it is entirely written in Java and more it has a Eclipse plugin that allows developing Jason projects under such IDE.

Let's now introduce another technology that was crucial for the development of our MAS, not only from a practical point of view but also from a more conceptual and philosophical one: CartAgo. It is a Java-written framework backgrounded with the Agents & Artifacts well-known theory and meta-model providing also a run-time platform for artifacts creation and execution. Moreover it has been perfectly integrated with Jason, so one more reason to adopt it.

The main motivations to choose CartAgo for artifacts management can be found in the theory behind it: we needed a mean to shape agents' environment and to give agents new capabilities upon needing. But there is another important feature more related to CartAgo artifacts implementation: every artifact embeds a monitor, so it allows for concurrent-safe agents execution and operation. Surely the reader knows how such feature may be crucial in a distributed MAS...

To end the chapter we briefly discuss other technologies used but related to the data mining features of ReadIT4mE!: MAUI and iText libraries.

MAUI - it is a Weka-based library that allows for automatic tagging, key-phrase extraction and even topic-assignment. It is very extensible and customizable, more it is completely open-source (as well as Weka in fact);

iText - this is a simple text management library needed to extract plain text from pdfs, html or else. It became very important because quite all the text mining tools are conceived to work only on plain text files.

2 System architecture

Here is a UML-style diagram showing Agents & Artifacts living and shaping the ReadIT4mE! MAS: the arrows are just a brief description of how these two entities are related to each other. The chapter A&A Interactions exhaust the argument in details.

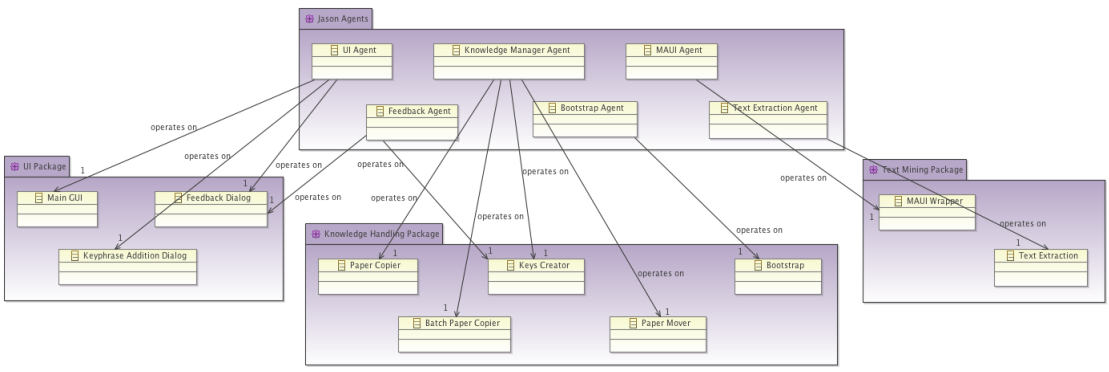

Figura 1.1: Class Diagram of ReadIT4mE!

Now let's zoom on every different package of the system to ease reading:

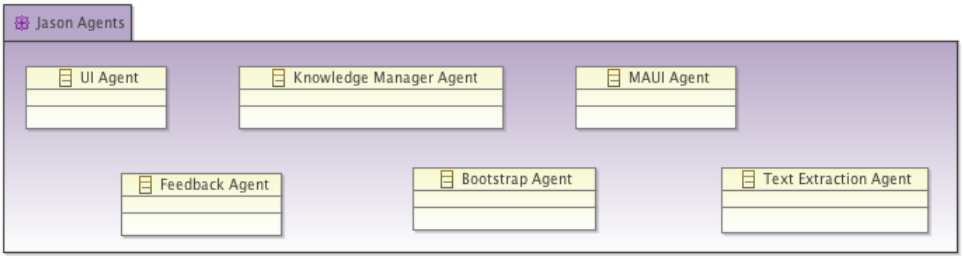

Figura 1.2: Class Diagram for Jason Agents

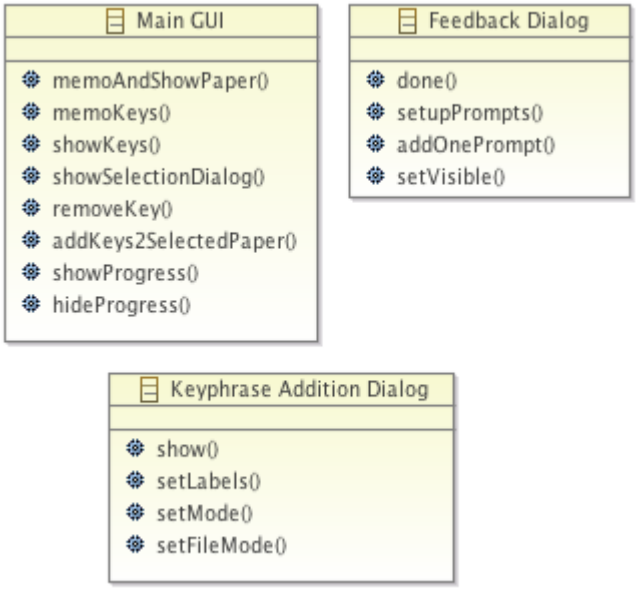

Figura 1.3: Class Diagram for UI package

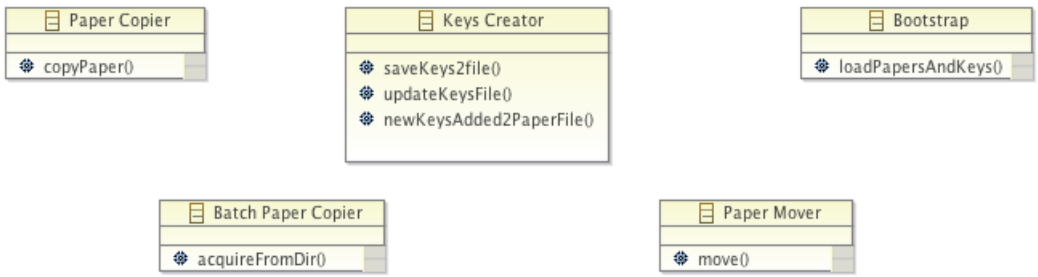

Figura 1.4: Class Diagram for Knowledge Handling package

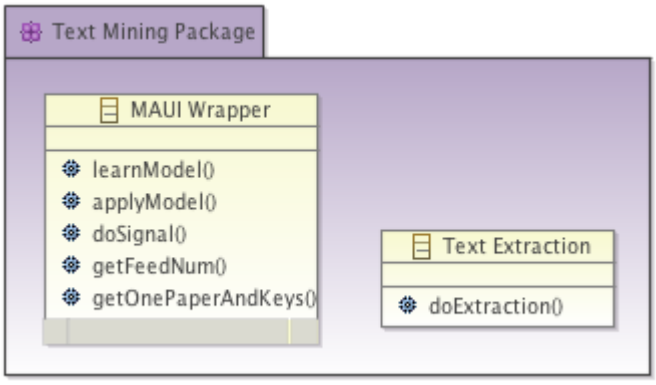

Figura 1.5: Class Diagram Text Mining package

Capitolo 2

Jason Agents description

In this chapter we describe the structure and behaviour of the different Jason agents living within our MAS. We try to do this from a logical point of view, hence describing what the aim of each agent is and how it tries to achieve such goal. The code snippets highlighted have only the purpose to better explain agents' mechanics: the great readability of Jason agents' code helps a lot achieving this.

1 UI Agent

As the name suggests, the UI Agent (uiAgent.asl code) is responsible for all the things regarding User Interface, hence it is in charge of managing the system GUIs and all the events generated by the user interaction with the system.

In the very end, UI Agent's reasoning cycle could be shortened in 3 very simple steps (repeated in loop):

1. setup tools - the agent builds its own artifacts and looks for the ones it needs from other agents;
2. perceive events - the agent waits for incoming GUI-generated events, reified in the form of beliefs added to its belief base;
3. react and show changes - the agent reacts to such events making decisions on how to modify the GUI according to user's expectations.

In the following picture we can see all the relevant events generated by the different components of the main GUI of the system:

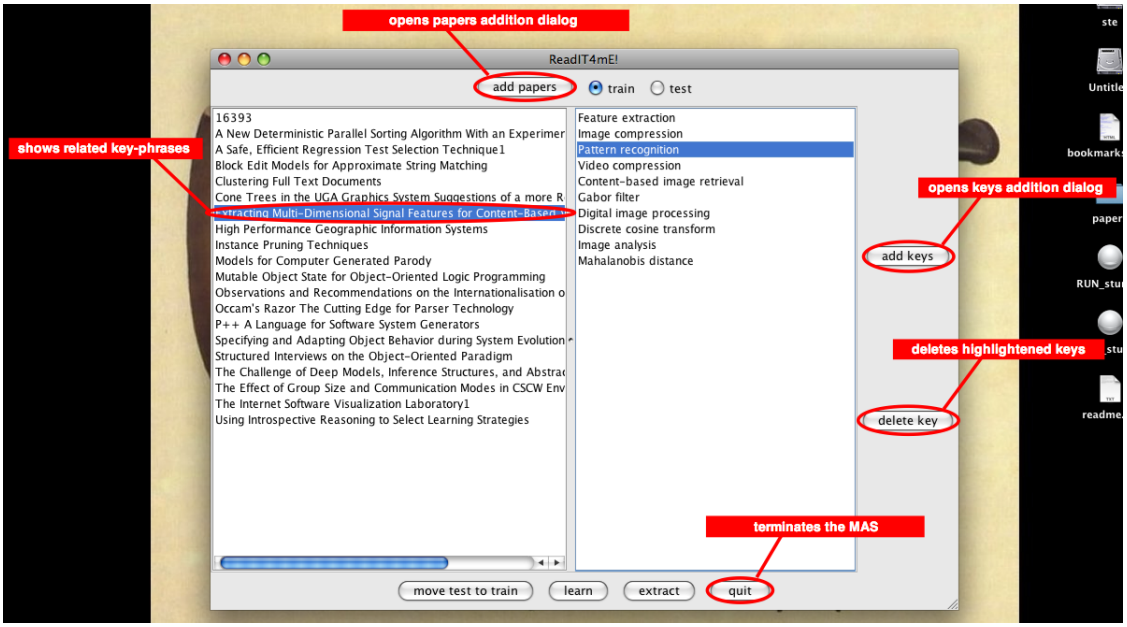

Figura 2.1: main GUI

opens papers addition dialog - when the corresponding button is pressed the UI Agent displays a proper dialog to let the user choose wich papers to add to the system. The Jason code is:

```
+openPaperDialog : true <- showSelectionDialog.
```

shows related key-phrases - when one of the paper in the list on the left is selected the UI Agent shows in the list on the right the related key-phrases. In the following Jason code the uppercase word is a Prolog-like variabile unified with the selected paper:

```
+paperSelected(PAPER) : true <- showKeys(PAPER).
```

opens keys addition dialog - upon button push the UI Agent displays a dialog to let the user enter new key-phrases for the selected paper. As before, the uppercase variabile is useful to associate the selected paper with the user-entered keys:

```
+addKeys(PAPER, MODE) : true <- !showKeysDialog(PAPER, MODE, true).
```

deletes highlightened key - when the button is pressed the UI Agent deletes the selected key from the corresponding paper's key-phrases list. The Jason code below follows previous notes:

```
+deleteKey(PAPER, KEY, _) : true <- removeKey(PAPER, KEY).
```

terminates the MAS - upon button push the UI Agent dispose the GUI and release related resources. Jason code here is really clear and concise:

```
+quit : true <- .stopMAS.
```

Following the typical flow of interactions between the end user and the system, next picture shows the paper addition dialog and what the UI Agent has to care of:

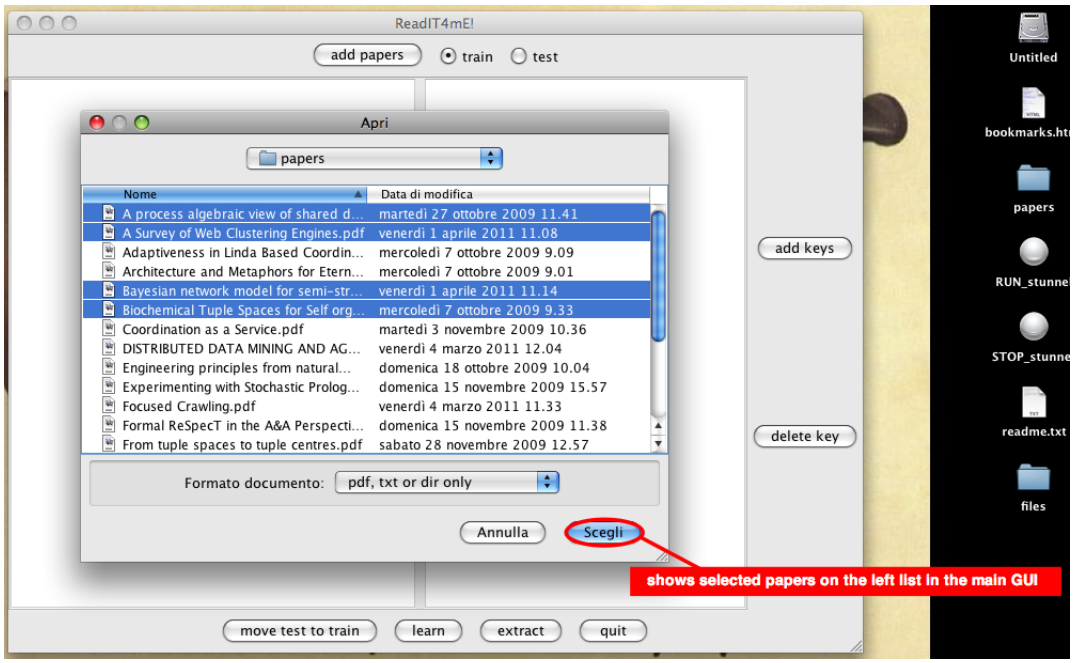

Figura 2.2: papers addition dialog

The only interesting event is the pression of the confirmation button highlighted in the above picture: when this happens the UI agent shows the new added papers on the left list in the main GUI and, if necessary, displays the key-phrases addition dialog (simply, for train-purpose papers keys have

to be supplied, while for test papers the user wants such keys back from the system):

```
+newPaper(PAPER, "train") [artifact_name(ID, "gui")] : true <-

     memoAndShowPaper(PAPER, false);
     showKeysDialog(PAPER, "train", false).

+newPaper(PAPER, "test") [artifact_name(ID, "gui")] : true <-

     memoAndShowPaper(PAPER, false).

+newPaper(PAPER, MODE) [artifact_name(ID, "bpc")] : true <-

     memoAndShowPaper(PAPER, false).
```

The above snippet of Jason code may seem redundant because of the repeated reactions to the same events, but this is the power of Jason engine: the square-bracketed text allows Jason engine to distinguish who is the source of each event. Combining this feature with the Prolog unification results in a endlessly powerful and flexible way to manage artifact-generated signals. In the A&A Interactions chapter we will cover more on this topic.

When the user choose wich papers to add to the initial training pool, another window pops up asking for correspondant key-phrases:

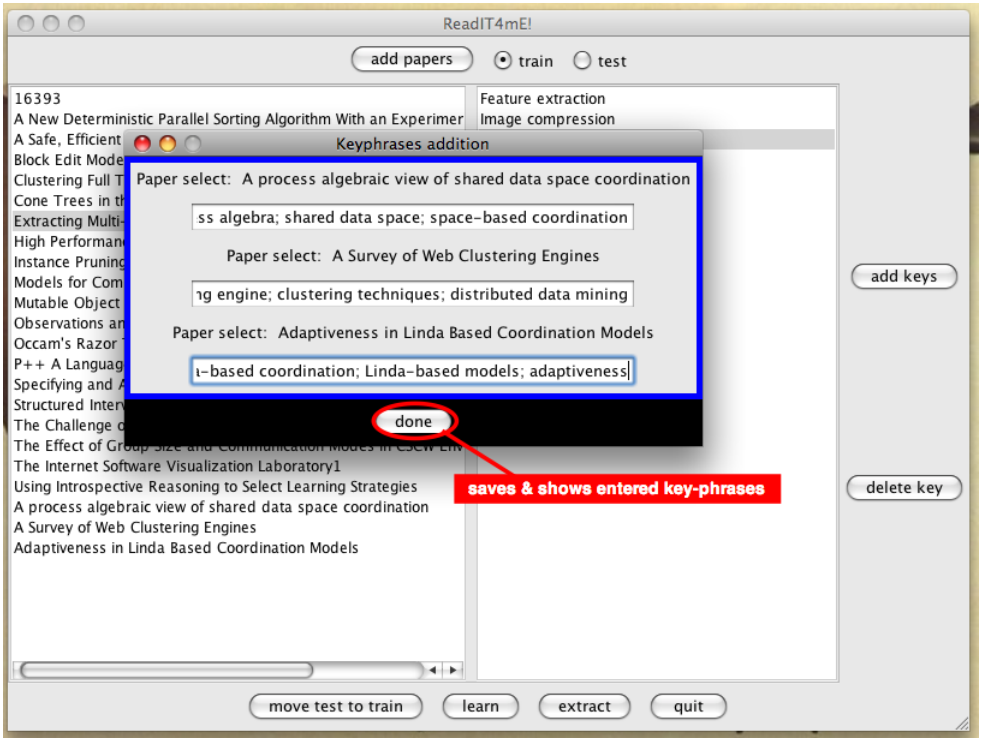

Figura 2.3: key-phrases addition dialog

Upon pressure of the unique button of the dialog the UI Agent takes the entered key-phrases and memorize them to later show 'em on the right side list whenever the related paper is selected:

```
+newKeysAdded2Paper(PAPER, KEY, _, EXISTS) : true <-

    addKeys2SelectedPaper(PAPER, KEY, EXISTS).
```

Don't mind the last Prolog-like variable wich is only for technical issues. The former two variables are quite self-explanatory.

The last screenshot we propose is about the last step of interaction between the user and the system, hence the feedback prompt. Once that the user runs the key-phrases extraction engine (the MAUI system) the system asks to him if he agrees with the extracted key-phrases or if he wish to discard some of them:

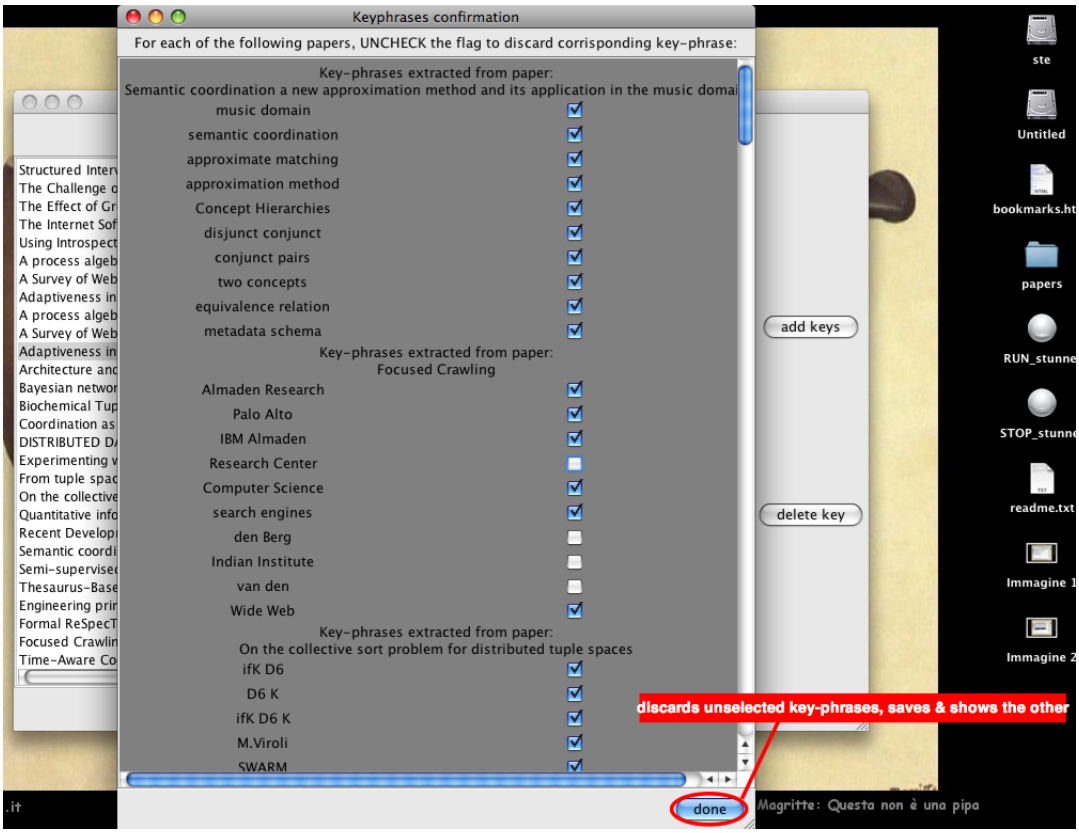

Figura 2.4: feedback prompt dialog

As before there is only one button in this dialog: upon push the UI Agent discards the unchecked key-phrases keeping (and thus showing) just the checked ones. To achieve this, the same event as the one seen in the main GUI generated by the button for key-phrases deletion is issued one time for each unchecked key-phrase:

```
+deleteKey(PAPER, KEY, _) : true <- removeKey(PAPER, KEY).
```

The reader may note that we haven't explained yet how such window is generated. Until now all the dialogs were UI Agent generated (the main GUI too, at the bootstrap of the system) in response to some event such as the pression of a button. The last window seen is different, because it is obviously generated by the UI Agent but in response to an event generated by the key-phrases extraction engine: we will come back to this point in the A&A Interaction chapter.

2 Knowledge Manager Agent

The Knowledge Manager Agent (knowledgeManagerAgent.asl) is responsible for all the issues related to filesystem management. Its behaviour is similar to the loop depicted for the UI Agent, so as done before we describe all the events interesting for the Knowledge Manager Agent in the following pictures:

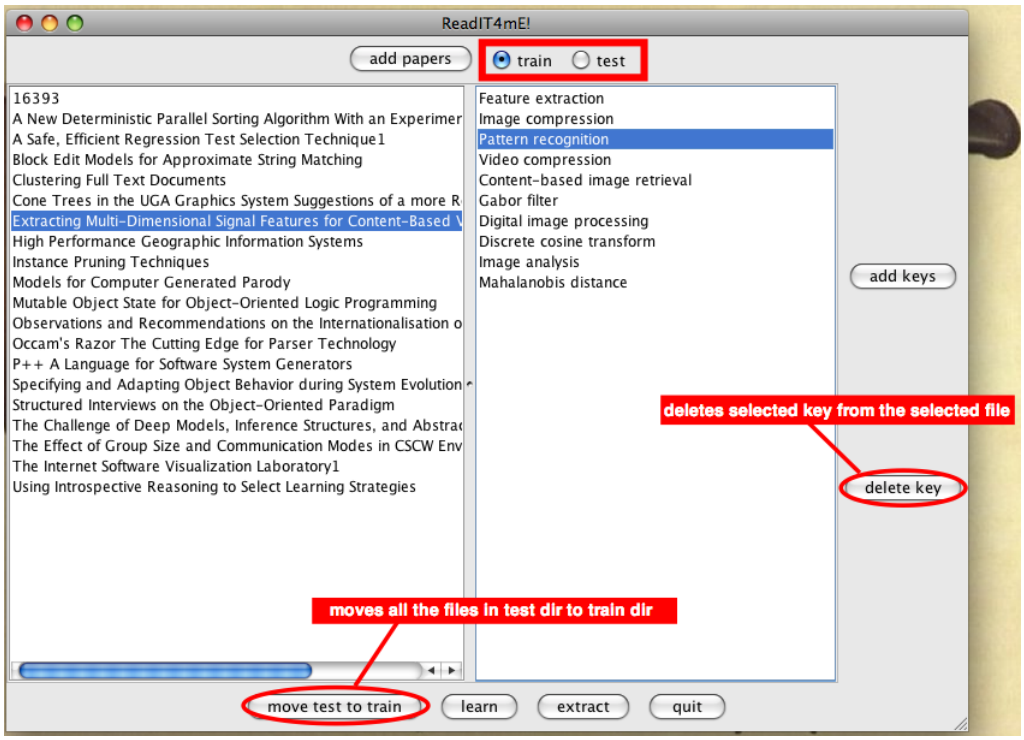

Figura 2.5: mainGUI

moves all the files in test dir to drain dir - upon button push the agent takes all the papers stored in the test directory and moves (hence copies than deletes) them to the train directory. This should allow the end user to re-train the text mining engine when necessary (namely when many papers has been successfully extracted, so the user wants an updated learning model for the engine).

```
+moveTest2Train : true <- move.
```

deletes selected key from the selected file - when this button is pressed the agent searches the project's internal storage directories for the

selected paper, takes the correspondant key-phrases storage file and deletes the selected key. The highlighted pair of radio button discriminates the directory in wich the selected paper is stored (it is used also for paper addition, see later on).

```
+deleteKey(PAPER, KEY, MODE) :true <- updateKeysFile(PAPER, KEY, MODE).
```

The last argument in the above snippet of code represents the checked radio button (the former two are quite self-explanatory).

The next picture shows the paper addition dialog, for wich there is only one relevant event:

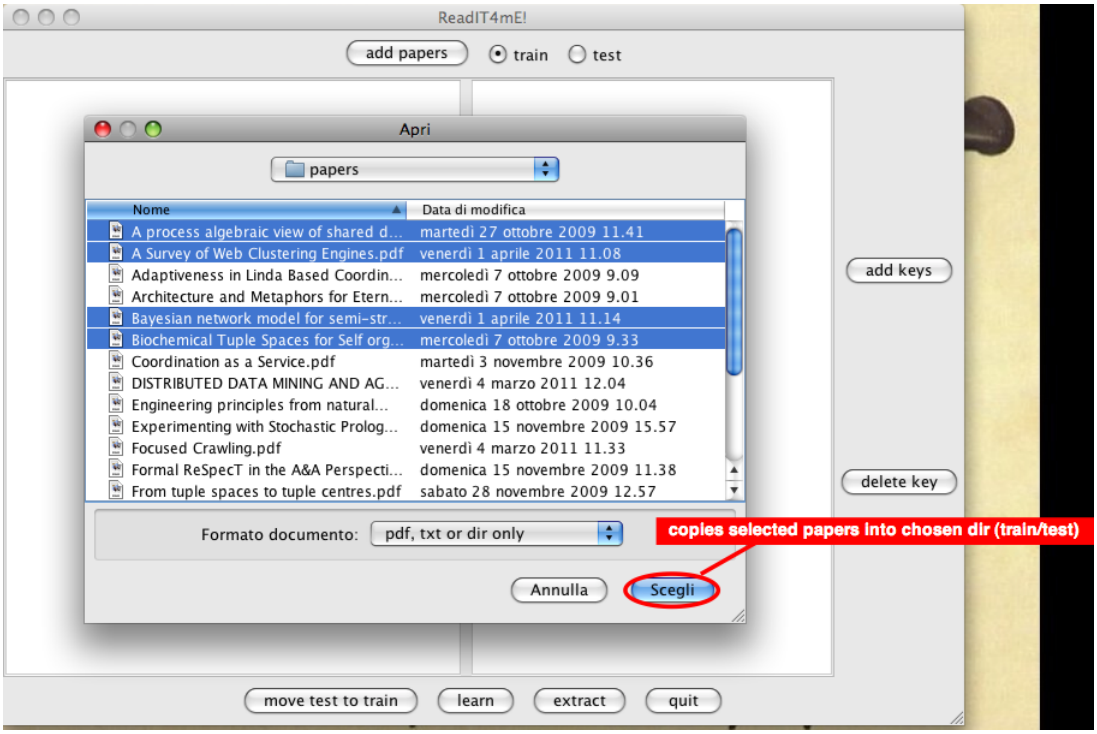

Figura 2.6: papers addition dialog

When the confirmation button is pressed, the Knowledge Manager Agent copies the selected paper in the proper directory (the one corresponding to the checked radio button, remember?) and if necessary it delegates the operation of plain text extraction ('cause they are in .pdf format almost always) to another agent of the system: the Text Extraction Agent (txtExtractorAgent.asl - not detailed here due to its simple algorithmic nature).

```
+newPaper(PAPER, MODE) : true <- copyPaper(PAPER, MODE).

+newExtractionCandidate(PAPER,MODE) : true <-

      .send(txtExtractorAgent, tell,
      newExtractionCandidate(PAPER, MODE)).
```

It is worth noting the usage of the Jason standard library method *.send* called in dot-notation: it allows inter-agent communication both of beliefs and goals (in this case, due to the keyword *tell*, we communicate a belief).

After papers addition, if and only if *train* radio button was selected, the system prompts the user to enter the keys related to those papers (in the *test* case there is no need for keys obviously...) with the already seen dialog:

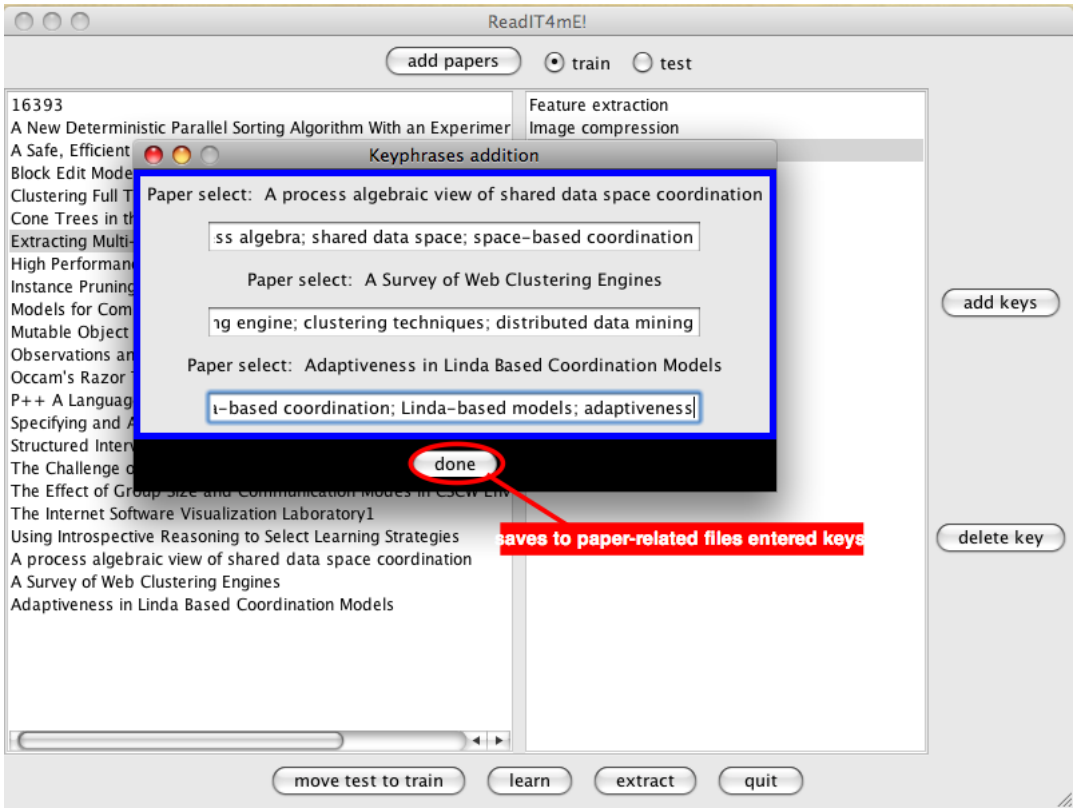

Figura 2.7: key-phrases addition dialog

When the highlighted button is pressed the agent creates a new file with the same name as the paper, one for each added papers, but extension *.key*

(necessary for the text mining tool used) in wich to store the entered keys one-per-line (they have to be comma separated, but white spaces don't matter...):

```
+newKeys(PAPER, KEYS, MODE) : true <- saveKeys2file(PAPER, KEYS, MODE).

+newKeysAdded2Paper(PAPER, KEY, MODE, EXISTS) : true <-
    newKeysAdded2PaperFile(PAPER, KEY, MODE, EXISTS).
```

The reason behind the existence of two different events is that the same dialog (the one above) is used both for first-time key-phrases prompt (when papers are added) and for later addition of key-phrases to existing papers. Having this in mind, the arguments' meaning is easily understable.

To end this section dedicated to the Knowledge Manager Agent we describe events generated by the feedback window:

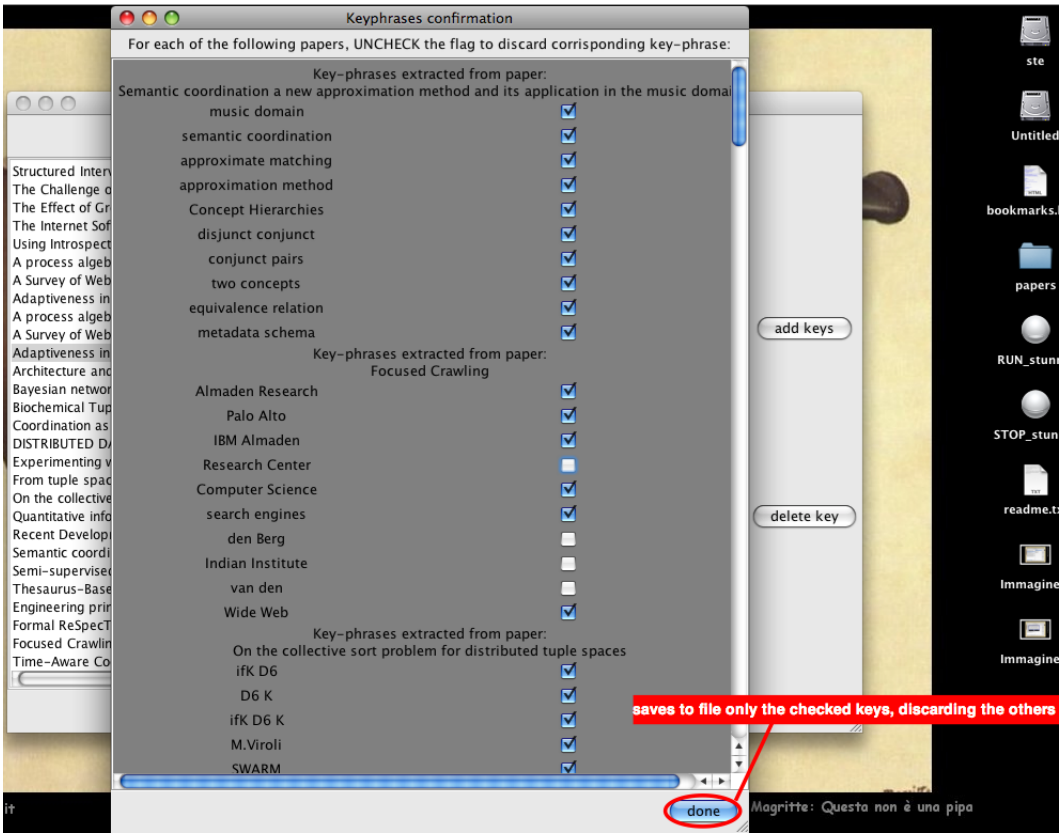

Figura 2.8: feedback prompt dialog

To tell the truth, there are no new events for the agent, because what we've done here for the sake of simplicity is to recycle the deletion event used before: it is generated for each unchecked key-phrase, so to allow the agent to retrieve the already existing keys-file (automatically created by the text mining tool) and delete the discarded keys.

```
+deleteKey(PAPER, KEY, MODE) :true <- updateKeysFile(PAPER, KEY, MODE).
```

3 Text Extraction, MAUI & Feedback Agents

We choose to describe these three agents all together because of their simplicity: in the very end they are just a bridge to the reach the desired functionalities given by system's artifacts (detailed in the next chapter). There is just one window that is source of some interesting events for these agents, the main GUI, while other events are generated elsewhere:

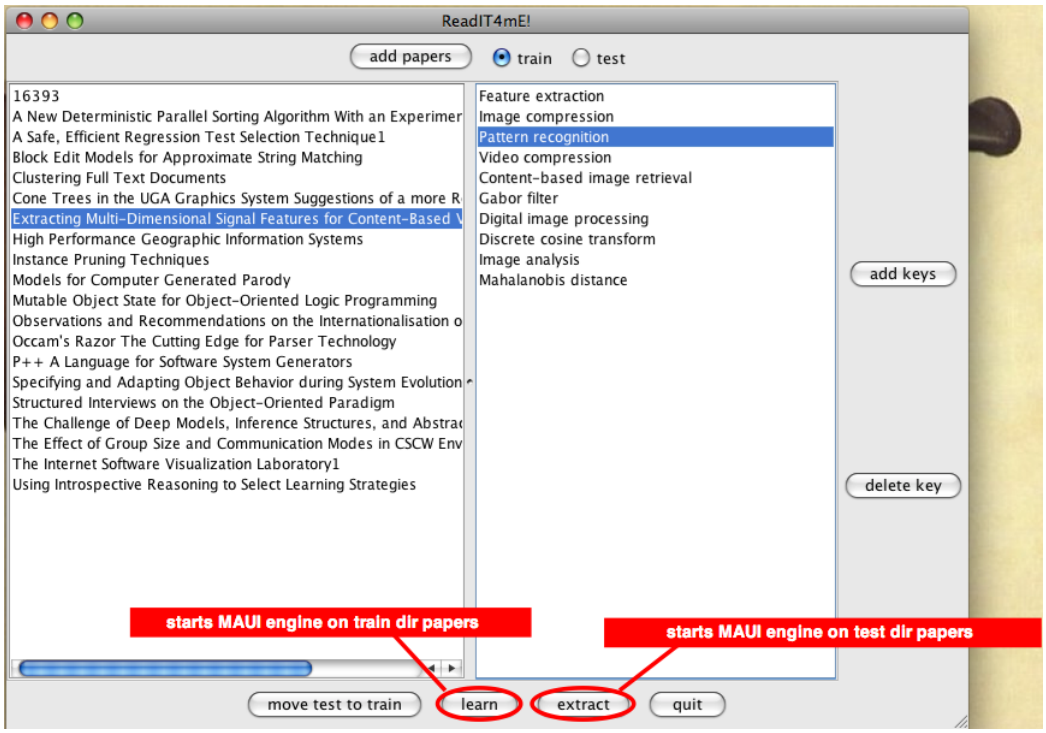

Figura 2.9: main GUI

In the Text Extraction Agent (txtExtractorAgent.asl) we can see both a simple example of reasoning and of inter-agent communication (as before

regarding beliefs): if the paper needs a text extraction step (namely it is in .pdf format) do it, else do nothing :) The two communications to the UI Agent are just because of the CPU-intensive extraction process, so we tell UI Agent to show a dialog telling the end user to wait:

```
+newExtractionCandidate(PAPER, MODE) : true <-

    if(.substring(".pdf", PAPER)){
    .send("uiAgent", tell, busy("started"));
    doExtraction(PAPER, MODE);
    .send("uiAgent", tell, busy("ended"));
    }.
```

The highlighted buttons are monitored by the MAUI Engine Agent (mauiAgent.asl) wich is responsible to manage the artifact that serves as a MAUI tool interface (see MauiWrapper.java in the Cartago Artifact description chapter):

starts MAUI engine on train dir papers - the corresponding button is used for training the text mining learning model inside MAUI tool with the papers in the train directory. Such papers must obviously have already an associated .key file.

```
+startTraining : true <- learnModel.
```

starts MAUI engine on test dir papers - is the typical usage of the system by the user, whenever he adds some paper and needs some keys back, automatically extracted by the MAUI engine.

```
+startTesting : true <- applyModel;

    doSignal.
```

For the moment do not mind the *doSignal* instruction: it is sufficient to say that it causes the visualization of the feedback dialog by the UI Agent through an event perceived by the Feedback Agent (feedbackAgent.asl) that acts as a bridge (see the A&A Interaction chapter for details):

```
+prompt4feedbacks : true <- .send(uiAgent, achieve, showFeedDialog).
```

It is worth noting that now the Jason standard library method *.send* has the *achieve* keyword in it (differently from the *tell* seen before): this way we are coomunicating to the UI Agent a new goal instead of a new belief.

4 Bootstrap Agent

A very dumb agent used for system start-up: in fact, we have to remind what was the state of the system at shut-down time, hence wich papers the user has added, where and with wich keys (if any):

```
+!loadKnowledge : true <- loadPapersAndKeys.
```

Such goal is communicated by the UI Agent with the pair *.send* - *achieve* as seen in previous section.

Capitolo 3

CartAgo artifacts description

It is now time to describe the means upon which the agents act, hence the Artifacts: they are tools forged and used by agents that enhances their capabilities and more, helps shaping the environment surrounding agents. CartAgo framework is entirely written in Java and basically we should get acquainted with some new notation and/or method:

@OPERATION - this tag must be pre-fixed to a method signature, and indicates that suc method is not a Java method but a CartAgo one, so not callable by Java class but only by Jason agents;

signal(...) - this is not the classical *signal()* used for Thread synchronization, but a new *signal()* that artifacts exploit to generate events observable by agents;

link*XXX*EventToOp(...) - this Java method is characteristic of GUI artifacts, a special kind of artifact used to efficiently manage GUI events. With such methods it is possible to link typical Swing-events to artifact special operations described below;

@INTERNAL_OPERATION - such operations are automatically called by the CartAgo platform when a GUI-event happens, allowing e.g. to signal something to interested agents observing the artifact (more on artifacts observation by agents is in the A&A Interaction chapter).

To allow a better understanding and to follow the clear separation of concerns wanted by the developers, we proceed from package to package.

1 UI artifacts package

1.1 Main Window Artifact

As the name suggests this artifact (*myMainWindow.java*) represents the graphical entry point of the ReadIT4mE! system, hence it is deeply connected to the UI Agent providing to it alle the means it needs to react to GUI events and modify it accordingly. Let's now first see a picture of it enriched with labels describing Swing-events generated and captured by the artifact itself:

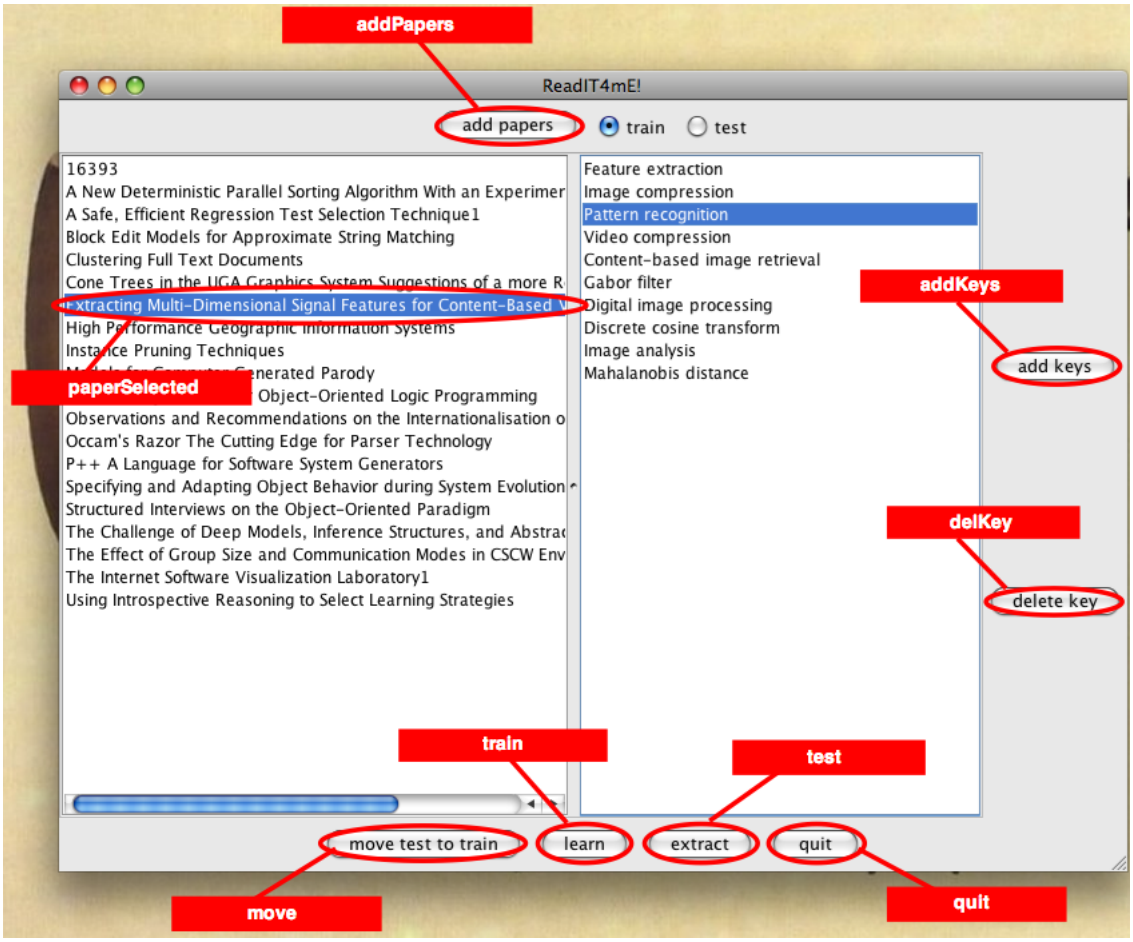

Figura 3.1: main GUI

addPapers - such event is linked to an *@INTERNAL_OPERATION* thanks to the above explained Java/CartAgo *linkActionEventToOp* and it al-

lows to generate an Agent-specific signal of kind *signal(openPaperDialog)*;

paperSelected - in the same way, this event generates a *signal(paperSelected, frmLittleResearchCommunity.list.getSelectedValue())*. We can now see how it is possible to embed into signals also values: the resulting Prolog-like term will have *paperSelected* as functor and the chosen value as the unique argument;

addKeys - generates *signal(addKeys, frmLittleResearchCommunity.list.getSelectedValue(), getSelectedRadioButton())*;

delKey - is source of *signal(deleteKey, frmLittleResearchCommunity.list.getSelectedValue(), frmLittleResearchCommunity.list_1.getSelectedValue(), getSelectedRadioButton())* signal;

move - has as a counterpart *signal(moveTest2Train)*;

train - linked to *signal(startTraining)*;

test - linked to *signal(startTesting)*;

quit - linked to *signal(quit)*.

To complete artifact's description we list below the methods' signatures provided by the Main Window Artifact to agents:

@OPERATION void memoAndShowPaper(...) - used by the UI Agent to show on the left list the names of the added papers;

@OPERATION void memoKeys(...) - used to internally store key-phrases on the right list (they will be displayed only when related paper is selected);

@OPERATION void showKeys(...) - to display such keys on the right list;

@OPERATION void showSelectionDialog() - to display the JFileChooser Swing component to allow user selects which papers to add;

@OPERATION void removeKey() - removes deleted keys from the internal storage medium (so that they will not be displayed anymore);

@OPERATION void addKeys2SelectedPaper() - another way to add keys to a paper (see the A&A Interaction chapter for an explanation of its usefulness;

@OPERATION void showProgress() - used by the UI Agent to show a pop-up dialog informing the end user to wait;

@OPERATION void hideProgress() - dually, it hides the progress window.

All the info related to how signals and operations are related and used by agents are delegated to the A&A Interaction chapter.

1.2 Key-phrase Addition Dialog Artifact

Another artifact almost exclusively used (but not exclusively observed...) by the UI Agent: the Key-phrase Addition Dialog Artifact (*MyKeyphraseMultiAddition.java*) is responsible for the keys addition step:

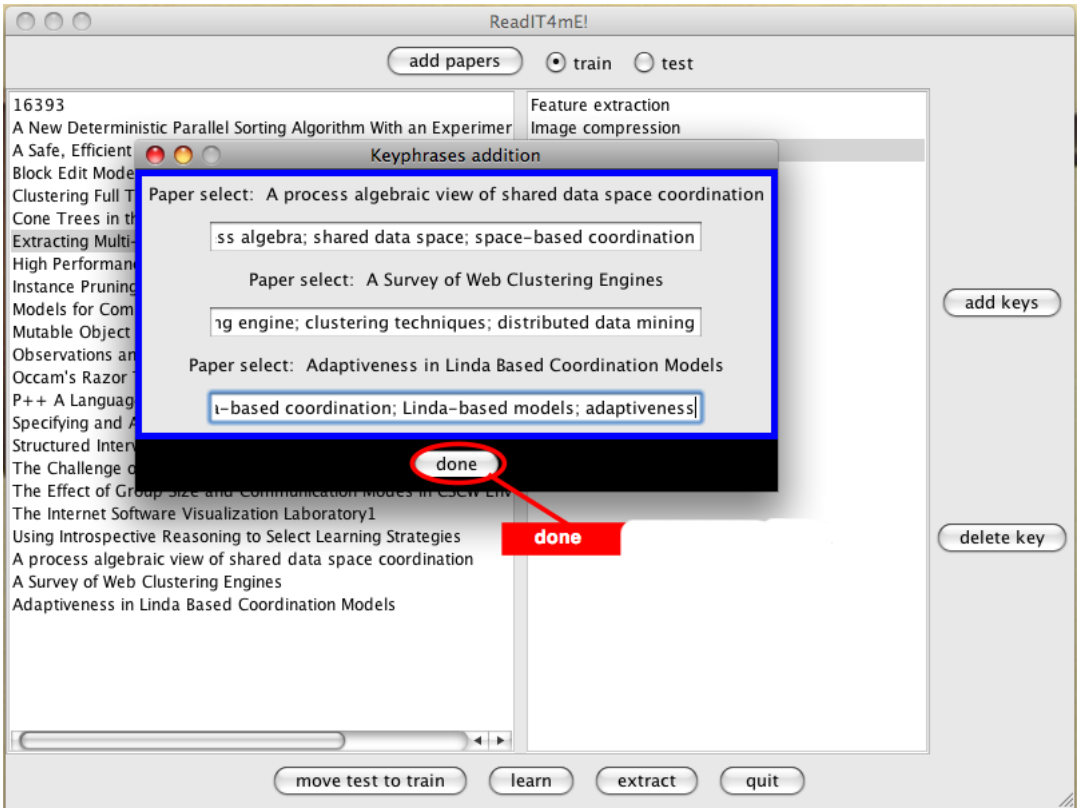

Figura 3.2: key-phrases addition dialog

Just a single event is interesting for observing agents:

done - generates *signal(newKeysAdded2Paper, temp.getPaper(), temp.getKeys(), mode, fileMode)* one for each added paper.

while two operations are made available (in truth they are four, but the other two are just for techincal purpose so we skip them):

@OPERATION void show() - used by the UI Agent to display the dialog upon need;

@OPERATION void setLabels(...) - used to properly set paper labels and allow user to better understand for which paper he is adding keys.

1.3 Feedback Dialog Artifact

To end the GUI-related artifacts overview we report the feedback window (*MyFeedDialog.java*):

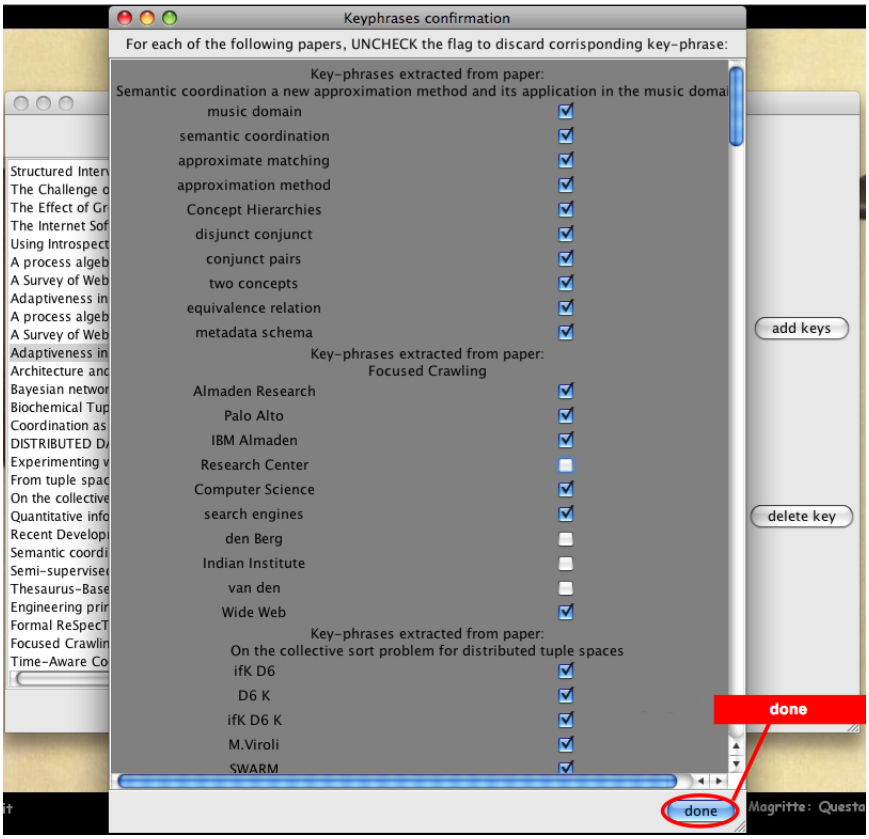

Figura 3.3: feedback prompt dialog

As before, we have just a single event is useful for the system:

done - generates *signal(deleteKey, pLabels[i].getText(), kLabels[(i*10)+j].getText(), test)* one for each unchecked key.

and two operations for agents (in truth there is another one, but just managing some view properties so not interesting):

@OPERATION void setVisible() - used by the UI Agent to display the dialog upon need;

addOnePrompt(...) - called for each paper upon which the extraction MAUI engine has worked to properly set the layout of the paper label, the key-labels related to it and the check boxes.

2 Knowledge Handling package

Here there are all the artifacts that have something to do with file management.

2.1 Paper Copier & Batch Paper Copier Artifacts

These two artifacts are grouped together because they are conceptually releated: both are used by the Knowledge Manager Agent to copy papers in the project's internal directories as soon as they are added to the ReadIT4mE! system, but while the former (*PaperCopier.java*) is exploited when the user explicitly selects one or more papers, the latter (*BatchPaperCopier.java*) is exploited when the user selects a directory, containing pairs of file paper-keys that the system has to add in batch mode.

Both have just one operation used externally by agents, althought they have also other methods used internally for different utility purposes:

@OPERATION void copyPaper(...) - provided by the Paper Copier, it copies the given paper in the project train/test directory (we already seen how this is discriminated) and generates the already seen event *signal(newExtractionCandidate, pathFile, mode)*;

@OPERATION void acquireFromDir(...) - provided by the Batch Paper Copier, simply generates two different events that delegates to other artifact the jobs to do. In particular:

- *signal(newPaper, filepath, mode)* - briefly, such event ends in the previous operation seen for the Paper Copier artifact, so to maximize reusability of artifacts and promote economy of signals flowing inside the system;
- *signal(newKeys, getPapernameFromPath(filepath, false), readSingleKeysFile(new File(filepath)), mode)* - this signal is used by another artifact that manages the input/output on file regarding key-phrases (see next section for its description).

2.2 Keys Creator Artifact

This is a quite complex artifact that manages all there is to do about key-phrases storage and retrieving, used exclusively by the Knowledge Manager Agent:

@OPERATION void saveKeys2file(...) - this operation is called when the user has entered the key-phrases in the related dialog and after pression of the confirmation button. Mind that this method is for first-time insertion case (keys file does not exists yet);

@OPERATION void newKeysAdded2PaperFile(...) - this is the already-existing file case, but the result of the call is the same as previous method;

@OPERATION void updateKeysFile(...) - this is called instead when keys are removed, not added: it searches for the selected key to discard inside the proper file and then updates it.

2.3 Paper Mover Artifact

In contrast, this artifact (*PaperMover.java*) is very simple and has just one method that generates two related signals directed toward the UI Agent:

move - this operation takes all the files in the test directory and moves them to the train directory. The purpose of such action is to allow the user to

periodically re-train the ReadIT4mE! system to continuously improve the text mining model used by MAUI. The two signals are meant to alert the user when the system is busy (with a lot of papers the moving process could take a while):

- *signal(busy, started)* - shows the progress pop-up dialog;
- *signal(busy, ended)* - hides the progress pop-up dialog.

2.4 Bootstrap Artifact

To end the section dedicated to file management, we describe the Bootstrap Artifact (*Bootstrap.java*) which is responsible for the boot-time loading of all the files needed by the system, such as papers and keys-files:

loadPapersAndKeys - to achieve its goal, this artifact generates two different signals that are perceived by the UI Agent so that it can make the GUI consistent with the files stored:

- *signal(loadedPaper, getPapernameFromPath(filepath, false)+;)* - for papers;
- *signal(loadedKeys, getPapernameFromPath(filepath, false), readSingleKeysFile(new File(filepath)))* - for key-phrases.

3 Text Mining package

Here we list the artifacts related to the MAUI tool.

3.1 MAUI Wrapper Artifact

This artifact represents an interface to the MAUI extraction tool: it is in fact a wrapper that allows agents to successfully exploit methods inside the MAUI jar. Describing the MAUI engine internal functioning is out from the scope of this paper, so it will not be done.

@OPERATION void learnModel() - this operation starts the MAUI learning engine, that analyzes the documents in input computing some text mining typical feature upon wich key-phrases selection is done. At the end of the process a model file is saved in the project directory named *knowledge*;

@OPERATION void applyModel() - this is the natural follower of the previous method because it applies the learned model to papers stored in the test directory. Once the extraction process is terminated, in the same directory the user can found the keys file automatically extracted;

@OPERATION void doSignal() - this is someway an accessory method, because it is used to signal to the other agents in the system that new keys have been found. More, it allows for feedback dialog showing by the UI Agent:

- *signal(newKeys, papersArray[i].toString(), keys, “)*;
- *signal(prompt4feedbacks).*

3.2 Text Extraction Artifact

The last artifact we highlight is a utility artifact that performs plain text extraction from the usually *.pdf* papers supplied. To do this it uses specific java libraries called *iText*:

@OPERATION void doExtraction(...) - a library method takes the given paper and extract plain text from it, saving the results in a *.txt* file with the same name.

Capitolo 4

A&A Interactions

This chapter is probably the most important of this report, because in here we describe in details the flow of knowledge between agents or among agents and artifacts: knowing which artifact is the source of a certain signal, which agents reacts to it and using which functionality of which artifact is extremely useful both to understand the complexity of the system but also how the system works.

To achieve the above described goal, we start from the single functionalities offered by the ReadIT4mE! system as already done before, following the user in each step of interaction with the GUI (we avoid a picture of the system GUIs because they all have been already seen before).

1 Paper addition step

The first step in the user interaction process is papers addition, for which he has two capabilities offered by ReadIT4mE!: single/multi papers selection and batch addition from directory. Whatever is the user choice, all the subsequent flow of info among agents and artifacts is caused by the pression of the *add papers* on the main GUI, that allows filesystem navigation.

Agents and artifacts involved in both cases are:

Agents :

- UI Agent;
- Knowledge Manager Agent;
- Text Extraction Agent;

Artifacts :

- Main GUI;
- Keyphrases Addition Dialog;
- Paper Copier;
- Batch Paper Copier;
- Keys Creator
- Text Extractor;

We choose to describe the two distinct cases in separate sections because flow of knowledge is different.

1.1 Single/multi selection

The artifact that knows first which papers the user has selected is the Main GUI, hence it is also the source of the signal generating the whole following computation: *signal(newPaper, filesSelected, getSelectedRadioButton())*. The first argument carries the list of the selected papers while the second keeps information about which is the directory where to put added files.

In the following picture the whole flow of interactions is described, along with the signals exchanged and the operations call done by agents on artifacts.

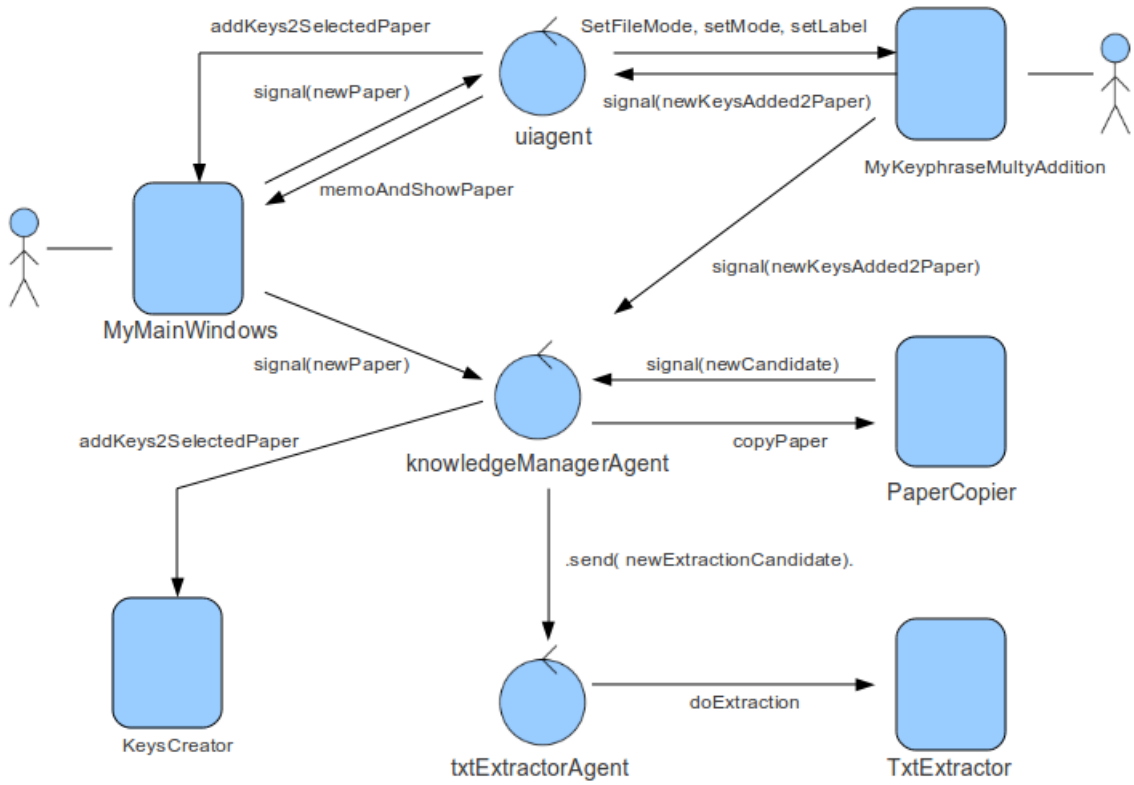

Figura 4.1: Add papers step - single/multi selection interactions

The above signal is perceived by the UI Agent and the Knowledge Manager Agent. The first one needs two different belief-addition plans to discriminate whether papers have been added to the train directory (hence we need user entered key-phrases) or to the test directory (thus there is no need for keys):

```
+newPaper(PAPER, "train") [artifact_name(ID, "gui")] : true <-

    memoAndShowPaper(PAPER, false);
    !showKeysDialog(PAPER, "train", false).

+newPaper(PAPER, "test") [artifact_name(ID, "gui")] : true <-

    memoAndShowPaper(PAPER, false)
```

We can easily see how in the first plan there is also a goal for Key-phrases Addition Dialog showing, while in the second not:

```
+!showKeysDialog(PAPER, MODE, EXISTS) : true <-
```

```
    !forgeKeysDialog(KSD, PAPER, MODE, EXISTS);
    focus(KSD).

+!forgeKeysDialog(KSD, PAPER, MODE, EXISTS) : true <-

    makeArtifact("ksd", "ui.MyKeyphraseMultyAddition", [], KSD);
    setFileMode(EXISTS);
    setMode(MODE);
    setLabels(PAPER).

-!forgeKeysDialog(KSD, PAPER, MODE, EXISTS) : true <-

    lookupArtifact("ksd", KSD);
    setFileMode(EXISTS);
    setMode(MODE);
    setLabels(PAPER);
    show.
```

We will come back on the keys addition step after some talk about the Knowledge Manager Agent. As we said before, he perceives the paper addition signal too, hence it has plans to properly react:

```
+newPaper(PAPER, MODE) : true <- copyPaper(PAPER, MODE).
```

The *copyPaper(...)* method is provided by the Paper Copier Artifact and generates *signal(newCandidate, pathFile, mode)* also useful for the Knowledge Manager Agent:

```
+newCandidate(PAPER,MODE) : true <-

    .send(txtExtractorAgent, tell, newExtractionCandidate(PAPER, MODE)).
```

We can see in the snippet of Jason code above a first usage of the communication capabilities possessed by agents: the Knowledge Manager tells to the Text Extractor there is anew paper candidate for plain text extraction. It will react with the following (already seen) plan:

```
+newExtractionCandidate(PAPER, MODE) : true <-
```

```
if(.substring(".pdf", PAPER)){
.send("uiAgent", tell, busy("started"));
doExtraction(PAPER, MODE);
.send("uiAgent", tell, busy("ended"));
};
```

To end the single/multi paper addition section we highlight in the following how the UI Agent and the Knowledge Manager Agent reacts to keyphrases insertion done by the end user. The *done* button push generates the event *signal(newKeysAdded2Paper, temp.getPaper(), temp.getKeys(), mode, fileMode)* observed by the two agents above:

UI Agent :

```
+newKeysAdded2Paper(PAPER, KEY, _, EXISTS) : true <-

    addKeys2SelectedPaper(PAPER, KEY, EXISTS).
```

Knowledge Manager Agent :

```
+newKeysAdded2Paper(PAPER, KEY, MODE, EXISTS) : true <-

    newKeysAdded2PaperFile(PAPER, KEY, MODE, EXISTS).
```

1.2 Batch addition from directory

The second case of paper addition is directory selection, which causes ReadIT4mE! to search the given path for *.pdf* or *.txt* papers and correspondant *.key* keys-files. The signal generated is *signal(newDirectory, selected[0].getPath(), getSelectedRadioButton())*: as before it informs the observing agents which is the directory to add papers (*train* or *test* upon *getSelectedRadioButton()*) and where to search for files with *selected[0].getPath()*.

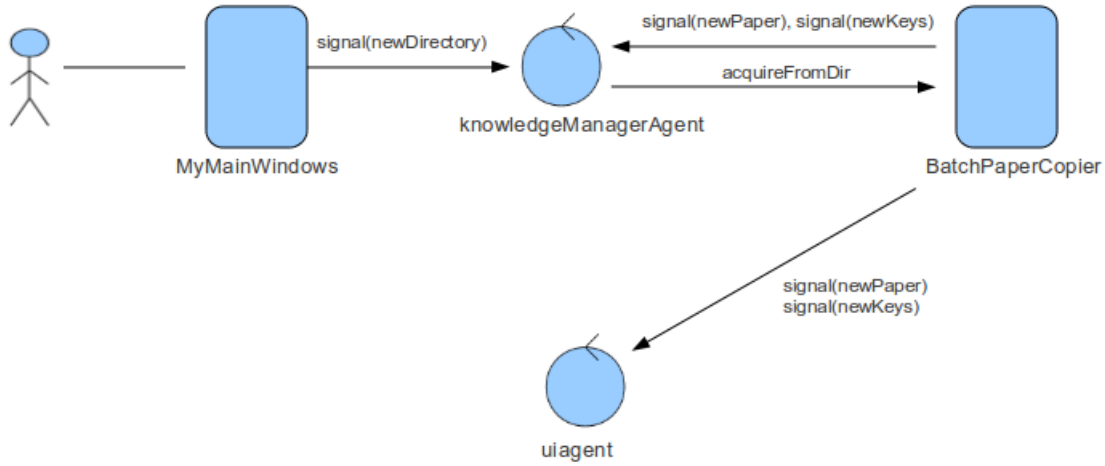

Figura 4.2: Add papers step - directory selection interactions

This time only the Knowledge Manager Agent will react to such signal, using an operation of the Batch Paper Copier Artifact that will in turn generate other signals: one for each paper found in the given directory and another for each keys file:

```
+newDirectory(DIR, MODE) : true <- acquireFromDir(DIR, MODE).

+newPaper(PAPER, MODE) [artifact_name(ID, "bpc")] : true <-

    memoAndShowPaper(PAPER, false).

+newKeys(PAPER, KEYS, MODE) : true <- saveKeys2file(PAPER, KEYS, MODE).
```

The last two signals above will be perceived also by the UI Agent, for which we have already seen the consequent behavior.

2 Training ReadIT4mE! step

After that the user is satisfied of its training pool of papers (with related key-phrases) he should start the training process over MAUI engine so to allow it to build its internal text mining model upon which later text extraction will be conducted.

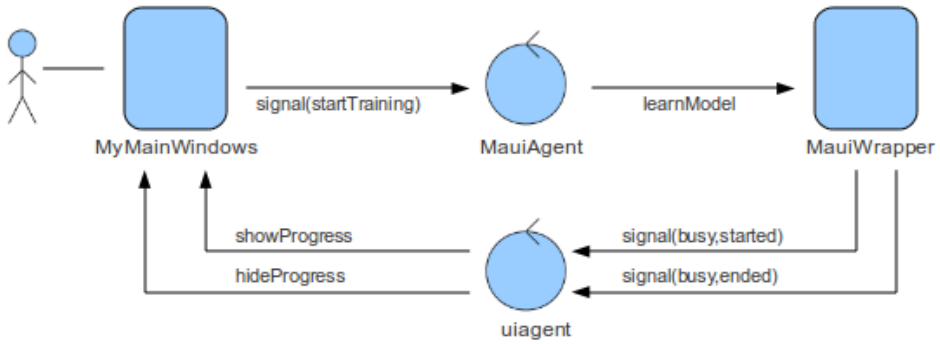

Figura 4.3: Training step

Agents and artifacts involved in the knowledge flow are:

Agents :

- MAUI Agent;

Artifacts :

- Main GUI;
- MAUI Wrapper.

Upon push if the *learn* button on the main GUI a *signal(startTraining)* is launched within the system, perceived by the MAUI Agent as follows:

```
+startTraining : true <- learnModel;
```

When the *learnModel* operations completes, a file named *myModel* is built in the project private directory *knowledge*: it the text mining model automatically learned from the training pool provided to MAUI.

3 Automatic key-phrases extraction step

Once that the MAUI model has been built the user can really start to use ReadIT4mE! completely: he can click on the *test* button to turn on MAUI extraction engine and obtain key-phrases for newly added papers that do not have them (namely, the ones stored in the *test* project directory).

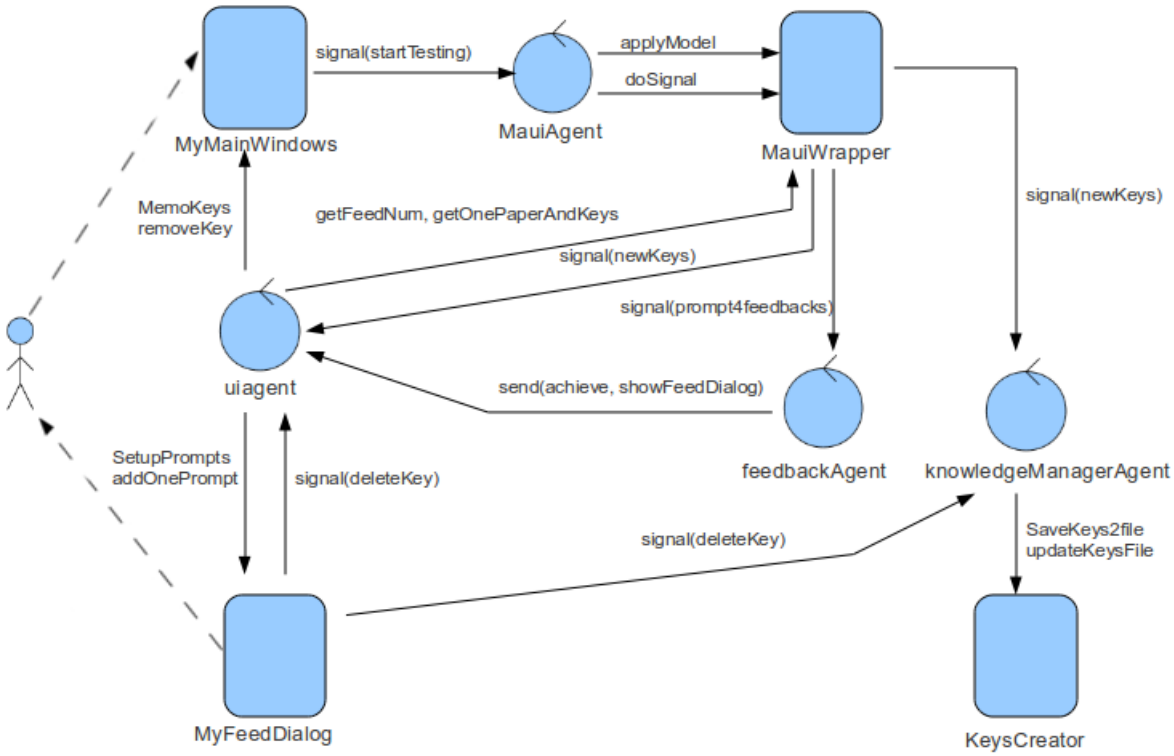

Figura 4.4: Testing step

This is for sure the most complex interactions scenario of the system, which involves several agents and artifacts (almost all):

Agents :

- MAUI Agent;
- UI Agent;
- Knowledge Manager Agent;
- Feedback Agent;

Artifacts :

- Main GUI;
- Maui Wrapper;
- Keys Creator;
- Feedback Dialog.

The signal responsible for the beginning of the whole flow of signals and operation calls is generated by the Main GUI artifact upon pression of the

correspondant button: *signal(startTesting)*.

The MAUI Agent only perceives such event and reacts starting the extraction process on the MAUI Wrapper artifact as depicted in the below snippet of Jason code:

```
+startTesting : true <-

    applyModel;
    doSignal;
```

The second operation call is crucial to the following interactions seen in the above picture: as its name suggests it is the source of important signals that are spread to the other involved agents. The first is *signal(newKeys, papersArray[i].toString(), keys, “)*, generated for each of the papers over which the MAUI engine has worked and extracted key-phrases; the second is *signal(prompt4feedbacks)*, launched with the aim to make the feedback dialog displayed.

The first signal highlighted above is perceived by two agents:

UI Agent - it simply store the keys internally for later visualization upon request (namely when te correspondant paper is selected from the left list of the main GUI):

```
+newKeys(PAPER, KEYS, _) : true <- memoKeys(PAPER, KEYS).
```

Knowledge Manager Agent - as already seen such agent stores the extracted keys in the appropriate directory (*train* or *test*):

```
+newKeys(PAPER, KEYS, MODE) : true <- saveKeys2file(PAPER, KEYS, MODE).
```

The second signal is instead managed only by the Feedback Agent, which does nothing more than to turn it to the UI Agent with the aim to dispaly the Feedback Dialog:

```
+prompt4feedbacks : true <-
```

```
    .send(uiAgent, achieve, showFeedDialog);

+!showFeedDialog : true <-

    makeArtifact("feedgui", "ui.MyFeedDialog", [], FD);
    focus(FD);
    !setupFeedGui;
    setVisible;

-!showFeedDialog : true <-

    lookupArtifact("feedgui", FD);
    !setupFeedGui;
    setVisible;
```

The *!setupFeedGui* goal is a plan useful to properly and coherently display the Feedback Dialog with the correct number of labels and check-boxes: it has no interests from a interaction viewpoint but is quite complex, so it will be skipped here.

Once that the Feedback GUI has been displayed, ReadIT4mE! waits that the end user pushes the *done* button to consequently manage discarded key-phrases deletion through the event *signal(deleteKey, pLabels[i].getText(), kLabels[(i*10)+j].getText(), test)*. Such signal is generated once for each of the unchecked key and is perceived by two different agents:

UI Agent - it simply reacts removing the discarded keys from the internal storage medium and refreshes the main GUI:

```
    +deleteKey(PAPER, KEY, _) : true <- removeKey(PAPER, KEY).
```

Knowledge Manager Agent - it has to update the already existing key-phrases file deleting the unchecked keys:

```
    +deleteKey(PAPER, KEY, MODE) :true <-

        updateKeysFile(PAPER, KEY, MODE).
```

4 Keys Addition/Deletion steps

In the following subsections we conclude the interactions chapter talking about keys addition and deletion. We will not list any figure because the beliefs and plans here described have been already seen in the chapters dedicated to the Agents and the Artifacts, moreover they are very simple.

4.1 Keys Addition

As we already know, the keys addition step may be cause by the user, pressing the *add keys* button on the main GUI, or auto-generated by ReadIT4mE! when papers are added in single/multi selection mode (and if and only if the *train* radio button is checked, as seen). In both cases the only involved agent is the UI, reacting to the previously described *signal(addKeys, ..., ...)* with the following plan:

```
+addKeys(PAPER, MODE) : true <-

    !showKeysDialog(PAPER, MODE, true).
```

4.2 Keys Deletion

This step is the same as the one seen when talking about the Feedback Dialog management, so we list just the generated event because the following plans actuated by the UI Agent and the Knowledge Agent have been previously seen.

```
signal("deleteKey",

    frmLittleResearchCommunity.list.getSelectedValue(),
    frmLittleResearchCommunity.list_1.getSelectedValue(),
    getSelectedRadioButton());
```

Future Works

The possible extensions for the ReadIT4mE! system are almost endless. Drawing features from the social networks and from the text mining fields the possibilities to enhance ReadIT4mE! are easily recognized to be an open-ended list of new functionalities:

- a first idea for further development may be to include in the system a Cluster Agent with the aim to provide the user with information about similarities between the different papers added to ReadIT4mE!. Obviously this agent may act on a simplicistic basis, e.g. marking as similar papers with a number of common keys greater than a minimum threshold, or on a more scientifically solid base, e.g. using a well-known clustering tools such as Weka (over wich MAUI itself has been conceived);
- another text mining oriented extension could be provided since now by the MAUI tool: it makes possible to validate extracted key-phrases comparing them with Wikipedia topics. This is done thanks to the Rapid Miner tool, embedded in the MAUI release.

A last further extension is more social networks oriented, and in truth was the very first ambition of the ReadIT4mE! system developers. In the very beginning we would like to give academic researchers not only a tool to avoid wasting time reading useless papers just to discover their content, but also to give them a medium to share their research together. We would like to make users of ReadIT4mE! connect to each other through the system itself, being alerted when some of their connected friends were adding new papers to their ReadIT4mE! systems and asking if they would like to obtain the same paper. This way ReadIT4mE! would become a mean to share knowledge and

automatically organize it on a semantic basis.

Such extension was later recognized to be very difficult to achieve with the frameworks and tools used for the exam, due to their novelty (above all CartAgo and TuCson). We trust that once new enhanced and stable release of such tools will be available, our planned sharing system will be feasible and very largely appreciated.

Or at least we hope so :)