UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique
UMR CNRS 7253 Heudiasyc

OMAS v10 - User Manual
Issue 3

Jean-Paul Barthès

BP 349 COMPIÈGNE
Tel +33 3 44 23 44 66
Fax +33 3 44 23 44 77

Email: barthes@utc.fr

N260
January 2013

Warning

This document contains the documentation for using the OMAS platform. It applies to the OMAS v9 release. However, some of the chapters were written with previous releases and may present some slight differences with OMAS v9. Main changes introduced in version 9 is a better handling of persistency, q different default window for Personal Assistant agents, easier functions for developing dialogs, and the mechanism of salient features to resolve in particular pronominal references. In addition OMAS can now appear as a FIPA compliant platform via the use of a FIPA transfer agent (postman) acting as a gateway. A chapter was added to explain this feature.

The OMAS platform was developed to simplify programming and facilitate building applications. However, it has a large number of features and the learning curve is rather steep.

Keywords

Multi-Agent Systems, Cognitive Agents

Revisions

Version	Date	Author	Remarks
1.0	Nov 10	Barthès	First Issue
1.1	Dec 10	Barthès	Changes in the Persistency chapter
1.2	Mar 11	Barthès	Updating the Personal Assistant chapter
			Section 4.8 added
			Section 5.15 added
	Jul 11	Barthès	Adding email and web mechanism for the PA
			Sections 5.16 and 5.17
1.3	Jan 12	Barthès	Upgrade for OMAS v8.2.0
2.0	Apr 12	Barthès	Upgrade for OMAS v9.0, FIPA compliant OMAS
2.1	Dec 12	Barthès	Inserting changes in OMAS vs JADE done by Marcio
3.0	Jan 13	Barthès	Inserting Web access for PAs

Contents

Foreword

OMAS is a platform for developing multi-agent systems, in which agents may be cognitive agents. It is intended for people who want to develop applications and not bother with the complex programming mechanisms inherent to multi-agent platforms. Thus, instead of providing a library of low-level modules allowing specialists to build a multi-agent platform, it offers a high level model of multi-agent systems. Thus, people wanting to develop simple applications can do so simply, and people wanting more control are given the necessary tools to exert it.

OMAS integrates the MOSS representation environment for expressing ontologies and knowledge bases. It was developed using MCL and ACL (Allegro Common Lisp), but due to the disappearance of Digitools and the evolution of the Macintosh OSX system, the Mac version has been temporarily suspended.

This document is a revised version of the previous manual and contains the documentation for developing OMAS applications. The first chapter indicates how to obtain and install OMAS, the following chapters are tutorials for beginners. The following chapters detail features of the OMAS platform. The MOSS representation language however has its own documentation. The last chapters deal with FIPA compliance, and a short comparison is made with JADE, a JAVA FIPA compliant platform.

Chapter 1

Installing OMAS

Contents

1.1 Environment Requirements

In order to run OMAS in the Windows environment (XP or Windows 7) one must first get a version of Allegro Common Lisp. Since the license is expensive, one can first use the free Allegro Express version that can be downloaded from the Franz web site. It is recommended to install the Lisp environment in the Program Files folder.

The Lisp environment can the be started by starting the executable "Allegro Common Lisp 9.0 (wIDE,ANSI)".

1.2 Obtaining OMAS

OMAS can be downloaded from the site http://www.utc.fr/∼barthes/OMAS. It is recommended to install the "OMAS-MOSS Release 10.0.x (ACL 9.0)" folder into the ACL folder.

Once the folders are copied to the right place, we can install OMAS.

1.3 Installing for the First Time

If the OMAS platform is installed for the first time then the following steps must be taken:

1. Start Allegro CL with IDE, go to the File menu and click the "File/compile and load ..." entry
2. Go to the OMAS-MOSS Release folder and select the "install.lisp" file The OMAS-MOSS platform will then load.
3. The rest should be automatic. For your information, it will create a startup.cl file in the ALLEGRO CL folder, which when you start the ACL next time over, will load OMAS automatically.

Note: if a startup.cl file exists in the ACL folder, ACL will start and execute whatever is in this file. Deleting this file allows starting in a clean Lisp environment.

1.4 Upgrading to a New Release

To upgrade from a running OMAS version to a new release, the following steps must be taken:

1. Load the new released version into the same folder as the current one.
2. Start ACL which should load current version of OMAS. If not, load current version of OMAS manually.
3. Go to the File menu, select the "compile and load ..." entry. Select the "upgrade.lisp" file in the OMAS-MOSS folder.
4. You will be asked to select the folder of the new released version.
5. You will be asked to allow clobbering the startup.cl file. Answer yes, unless you are using it for launching things other than OMAS.
6. The rest should be automatic. For your information, your MOSS and OMAS application files will be transferred to the new release, and the startup.cl file will be upgraded, so that next time you start Lisp the new version of OMAS is started automatically.
7. Exit.
8. Check that your application files in the applications folder have been transferred.
9. Restart ACL. It should boot on the new version and your applications folders should contain your applications. The sample-applications folder may contain examples different from the previous ones.

Note: The MOSS and OMAS folders named "applications" will be transferred (copied) to the new OMAS-MOSS Release, the startup.cl file will be recreated in the ACL folder, and the omas.properties file will be transferred so that current OMAS parameters are preserved.

1.5 The OMAS Configuration File

The OMAS folder contains a configuration file that is updated automatically, but can be modified manually. It contains parameters controlling some features of the OMAS environment. When loading OMAS for the first time the content of the file is the following:

```
;;; This file contains parameters for the OMAS multi-agent platform

SITE = THOR
COTERIE = NULL
BROADCAST =
PORT = 50000
AUTOMATIC = NIL
INTERACTION = NIL
CONTROL-PANEL = T
GRAPHICS-WINDOW = T

;;; EOF
```

The meaning of the various parameters is the following:

SITE	name of a local reference, e.g. THOR (a disk name) or UTC.
COTERIE	the name of the coterie or application
BROADCAST	the UDP broadcast IP of the LAN
PORT	the current port used for exchanging messages (default is 50000)
AUTOMATIC	if T, then the application will be loaded automatically, bypassing the init window
INTERACTION	if T, then a specific interaction window must be provided for a PA
CONTROL-PANEL	if T, make the control panel appear after agents are loaded default)
GRAPHICS-WINDOW	if T, makes the message diagrams window appear after agents are loaded

Chapter 2

OMAS Tutorials

Contents

This chapter contains three tutorials to help users get some feeling for the OMAS platform. The first tutorial is very simple and shows how to create an agent in the ACL debug window and how to use simple commands of the OMAS IDE. The second tutorial shows how to create a simple application in the application folder, using the default files. The third tutorial shows how to use more advanced features.

2.1 Tutorial 1: Simple Introduction to the OMAS Platform

Tutorial 1 shows how to launch the platform, how to create an agent, give it skills and use the IDE interface.

2.1.1 How to Get Started

First, one has to load the platform. To load the OMAS, after OMAS has been installed (for the first time) simply launch the LISP environment (Allegro CL with IDE). An initial window will appear as shown Fig.2.1 (ignore the various warning messages displayed while loading).

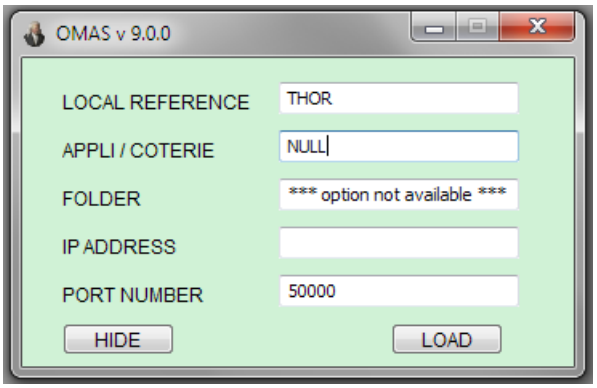

Figure 2.1: Initial OMAS init window (Windows 7)

Thereafter, clicking the LOAD button produces the OMAS Control Panel (Fig.2.2) on the top left part of the screen. The control panel allows controlling and tracing what is happening with the agents. A graphics trace window entitled "Message Diagrams" also appears at the top right part of the screen.

You can reshape the Debug window to see the control panel the graphics trace and ACL debug window at the same time (Fig.2.3).

2.1.2 Creating an Agent

Creating an agent is simple. Just type the following line into the listener window (Debug Window) shown Fig.2.4[1]:

```
CG-USER(1): (defagent :TEST)
#<AGENT TEST>
TEXT(2):
```

[1]To be able to type something into the Debug Window, one must click the top left button within this window, containing the symbol "> _" This triggers a prompt like CG-USER(1): meaning that you are in the Common Graphics User package (a name space called CG-USER), and this is interaction 1.

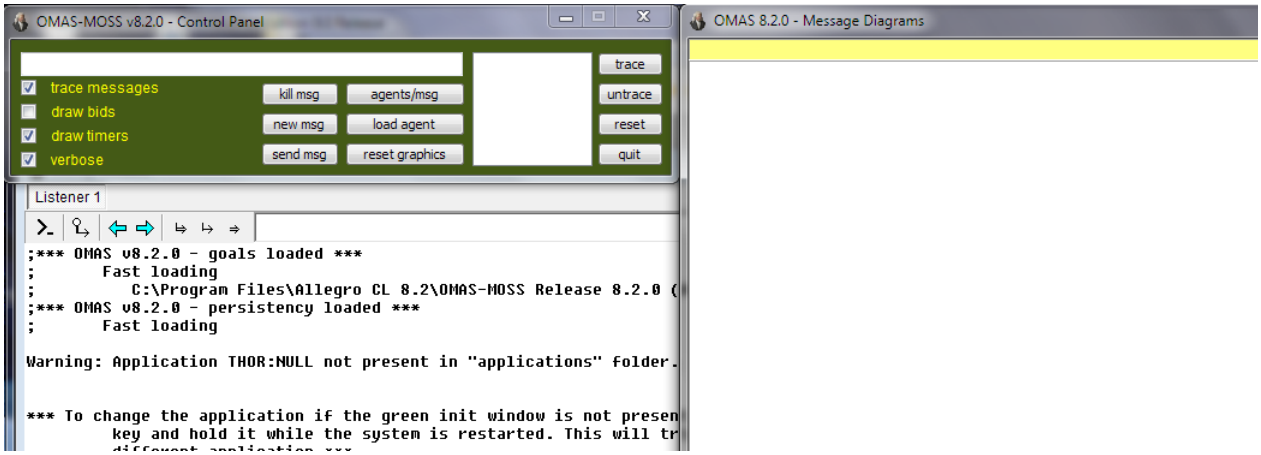

Figure 2.2: OMAS initial set up showing OMAS control panel on the left and graphics file on the right and the ACL listener (Debug pane) on the background (Windows 7).

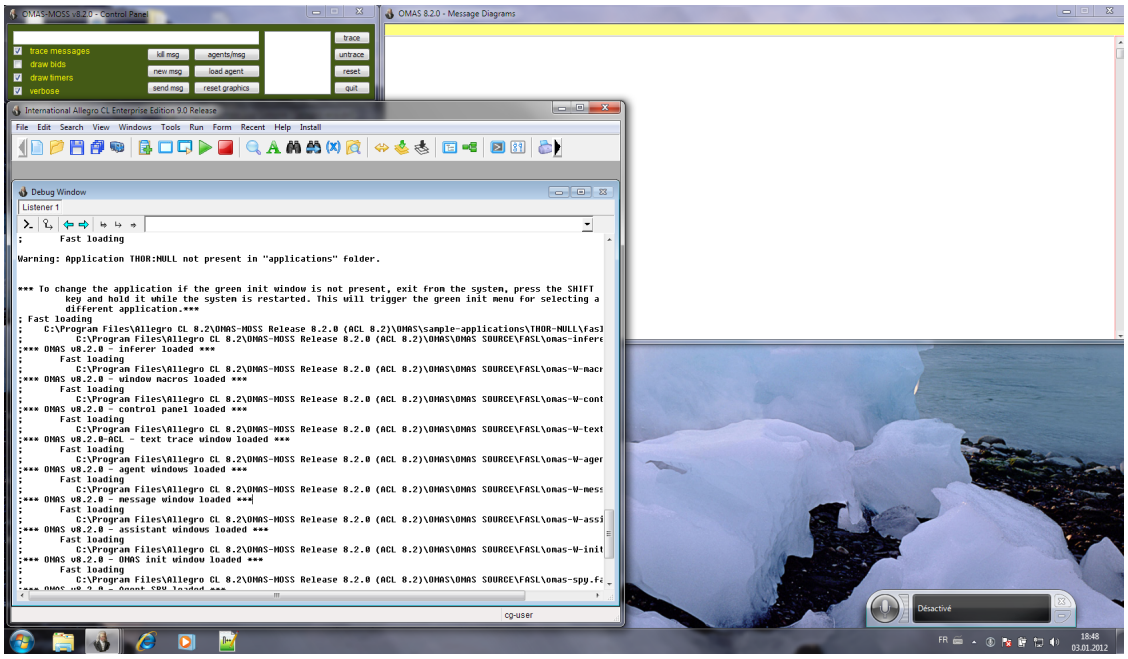

Figure 2.3: OMAS working environment (Windows 7).

The agent named `TEST`[2] has been created and is now running. Its name appears in the control panel as SA_AGENT[3]. Note that we use a keyword[4] for the name of the agent. The control panel is updated but not the message diagrams window. You can update it by clicking the "reset graphics" button of the control panel.

Note that the ACL prompt has been modified and now reads "TEXT(2):". Indeed, each agent is created in a protected name space or *package*. Lisp starts with a neutral environment called CG-USER, a nickname for COMMON-GRAHICS-USER. Creating the TEST agent also created the TEST

[2]Remember that LISP does not make any difference between cases, thus TEST, Test, test or TeSt represent the same object.

[3]The SA prefix means Service Agent.

[4]A Lisp keyword is a symbol prefixed with a column (:), e.g. :ALBERT.

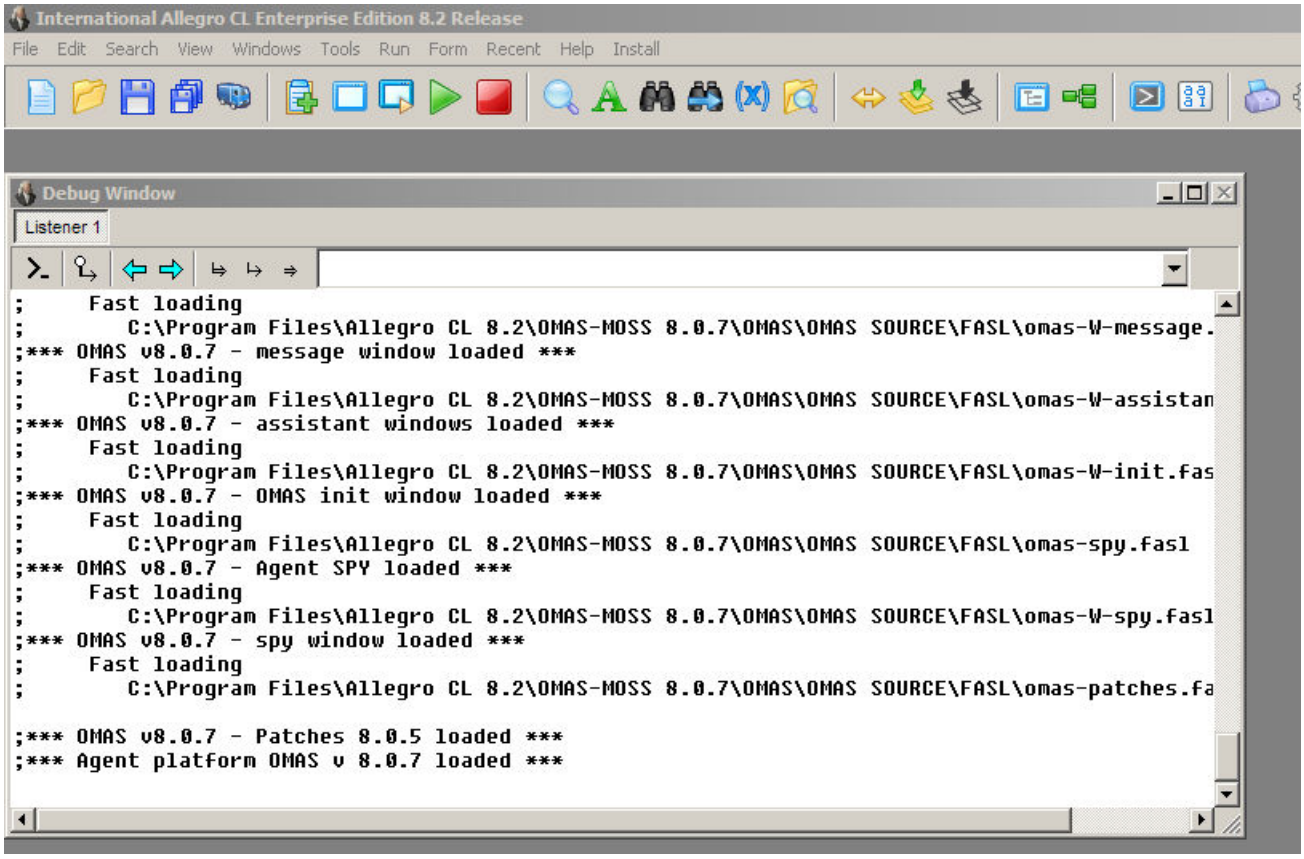

Figure 2.4: The Lisp Listener (ACL Debug Window) after OMAS is loaded (may not look exactly as this one)

package and switched to that package. To change from one package to another one can use one of the following commands:

```
(in-package :CG-USER)
```

or

```
:pa :cg-user
```

which will return to the CG-USER package.

One problem is that our agent TEST cannot do anything useful yet. To enable it to do something interesting, one must define skills.

2.1.3 Giving Skills to an Agent

A skill is implemented as a LISP function. To give a skill to an agent, one first must declare the skill and then write the function to handle it. For example let us give our agent the possibility to greet us when receiving a greeting message. This is done as follows:

```
TEST(2): (defskill :GREETINGS :TEST :static-fcn static-greetings)
(:GREETINGS :TEST)
```

Note that this must be done in the agent namespace (package). The agent has been given the skill `:greetings`; now me must write the function implementing the corresponding behavior.

```
CG-USER(3): (defun static-greetings (agent message arg)
    "function that returns a string to the sender"
    (static-exit agent "Greetings..."))
STATIC-GREETINGS
```

The two first arguments (agent and message) are required, the last one (arg) is the first argument of the message received by the agent; agent is the current agent TEST, message refers to the request message object that is received when the function is executed.

More details: Note the use of the `static-exit` function. It takes two arguments, the first one, `agent`, is required, the second one is what will be returned to the sender of a request message. A simple skill is said to be *static*. An agent can perform the corresponding task locally without relying on other agents, i.e. without sub-contracting.

Skills vs. methods: Since skills are implemented as functions, one could claim that there is no difference between a skill and a method. However, in the case of agents, when the agent possesses a skill and receives a message requiring to use the skill, it may choose not to answer (because it is busy, angry with the sender, overworked, tired, etc.). Within OMAS however, agents are benevolent and try to help, hence they usually (but not always) answer.

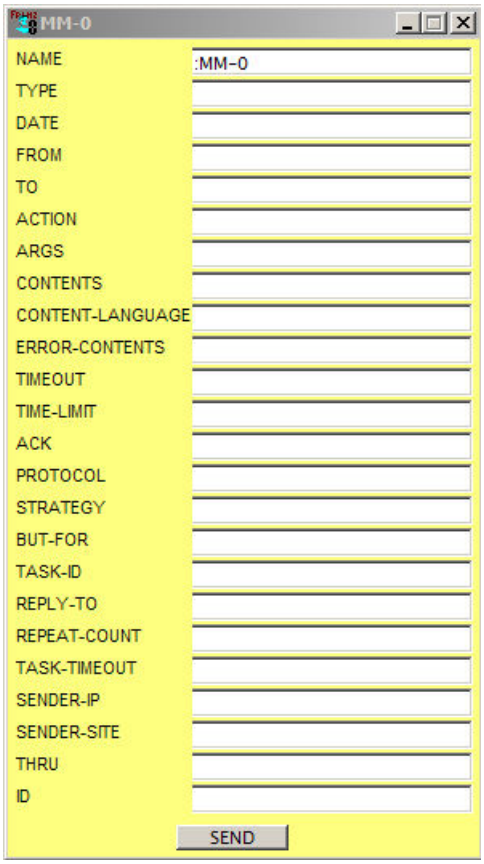

Figure 2.5: Message editing window

2.1.4 Exercising the Agent: Sending a Message

We can now exercise our agent by sending it a message. First we must create a message. To do so, we click the "new msg" button in the control panel. A message window appears as shown Fig.2.5.

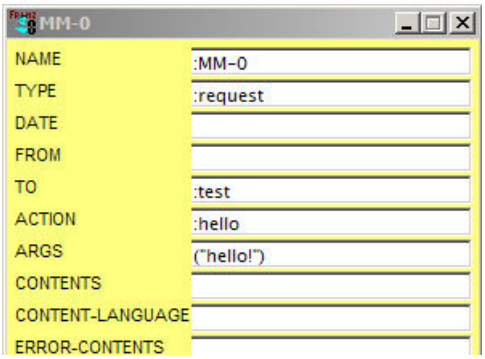

Figure 2.6: Filling the message window for the agent TEST

We fill some of the fields of the editing window, namely:

```
Type   :request
To :test
Action :greetings
Args ("Hello!")
```

We do not need to fill anything else.

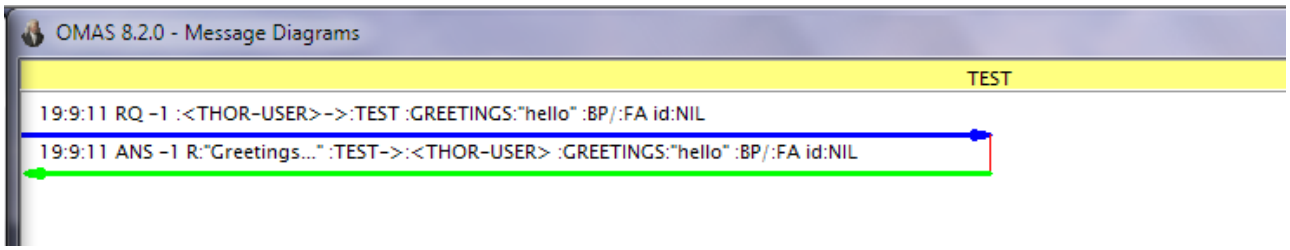

Figure 2.7: Trace of the exchange of messages in the Message Diagrams window

We then click the "send" button at the bottom the message is sent to TEST and TEST answers it. A trace is shown in the Message Diagrams window (Fig.2.7). Since the answer is to the system, the answer message is displayed in the small horizontal window of the control panel (Fig.2.8). It says:

```
21:12:39 ANS -2 R: "Greetings..." :TEST-> :GREETINGS:"Hello!" :BP/:FA
```

meaning that the message was returned at 21:12:39, was an ANSwer to task number -2, whose content is "Greetings?" and came from `:TEST` that used the skill `:GREETINGS` on the argument "Hello!"

It is also possible to print the content of the answer message into the IDE debug window by calling the user-get-last-answer function in the TEST package or in the CG-USER package as follows:

```
TEST(7): (user-get-last-answer)
"Greetings..."
```

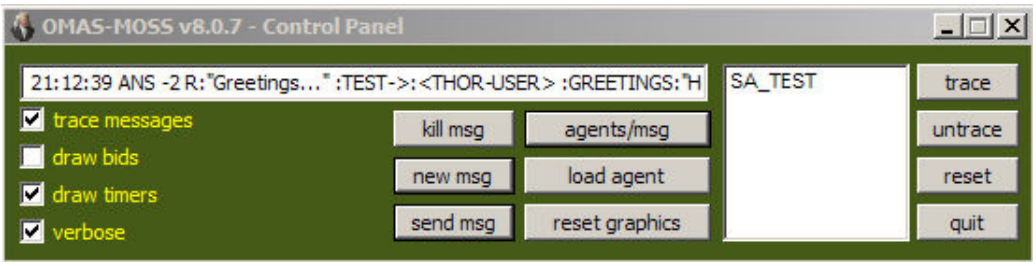

Figure 2.8: Panel showing returned message

```
TEST(8): :pa cg-user
CG-USER(9): (user-get-last-answer)
"Greetings..."
```

2.1.5 OMAS Control Panel

The panel has a number of features, associated with the various buttons.

Showing Agents or Messages

The "agents/msg" buttons commands the content of the pane where agents are displayed (i.e. where SA_TEST appears). If we click it, then the MM-0 message appears replacing SA_TEST (Fig.2.9). If we click it again the TEST agent reappears.

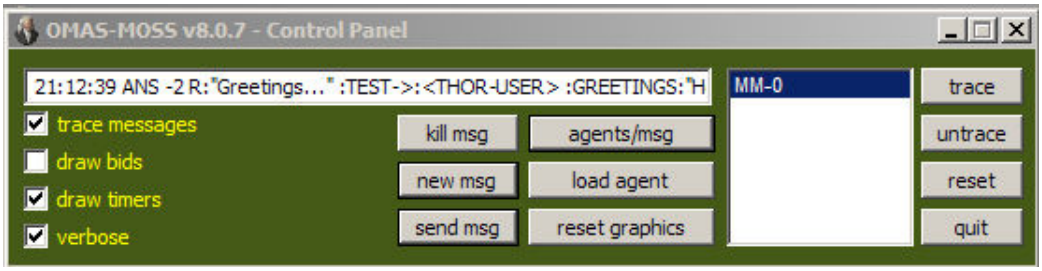

Figure 2.9: Displaying messages (MM-0) in the scrolling list pane

If we double-click on MM-0, then the edit window appears (Fig.2.10).

Note that some of the fields have been filled automatically with default values (date, from, time-limit, protocol, strategy).

Now if we double-click the agent SA_TEST, a special agent window appears right underneath the control panel as shown Fig.2.11. The content of this window will be detailed later.

Showing the Exchange of Messages

The exchange of messages is shown in the Message Diagrams window. The window can be refreshed by clicking the "reset-graphics" button. Let us refresh the window and send the message MM-0 again by selecting the message in the message list and clicking the "send msg" button. We obtain the same result as previously, two messages appearing in the message diagrams window (Fig.2.7): One sent to the TEST agent (blue request) and one returned by the agent (green answer).

Note that the messages come and return to the left hand side of the graphics window. This is where the user lives!

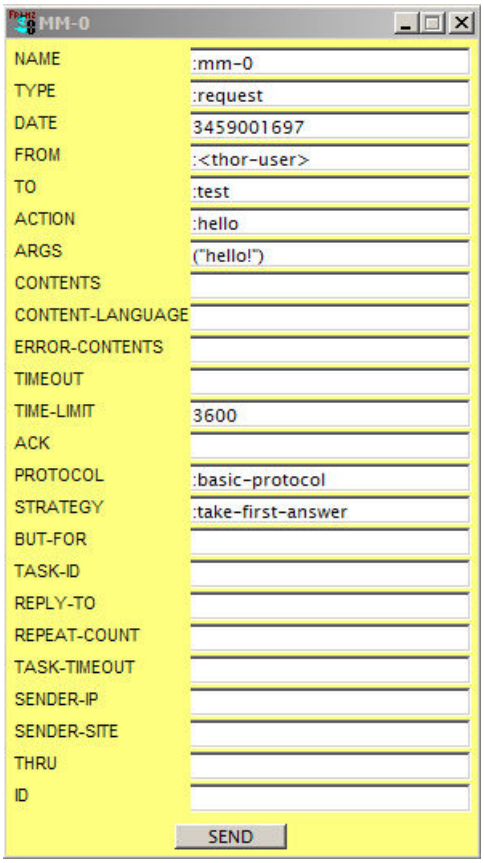

Figure 2.10: Message edit window

Figure 2.11: Agent window for TEST

Sending a Message Again

To send a message again, one can either double-click on the message, and click on the "send" button at the bottom of the message edit window, or select the message in the scrolling list and click the "send msg" button of the control panel.

New messages can be defined. They will appear in the scrolling list of the control panel. The system will give a new name, MM-xx, to each new message

Miscellaneous Control Buttons

Several control buttons appear on the control panel.

On the right-hand side:

Trace	Allows printing a trace to follow what an agent is doing in a special "Text Trace" window (the verbose option should be checked). The agent is selected from the scrolling list
Untrace	Cancel the trace for an agent
Reset	Resets the system by reinitializing the various processes. Currently disabled
Quit	Quits the application after confirmation

Checkboxes on the left hand-side:

Trace messages	Toggles the text on and off on the Message Diagram
Draw bids	Draws (or not) the bids when the Contract Net protocol is used
Draw timers	Draws the life-line for the timers, when checked
Verbose	Prints a detailed trace of what is happening when checked

Buttons on the left hand-side:

Kill msg	Removes the selected message from te list of messages
New msg	Calls the message window to define a new message
Send msg	Sends the selected message from the list of messages
Agent/Message	Toggles the list of agents and messages
Load agent	Manually load an agent from its files (advanced)
Reset graphics	Clears the graphics window of all the printed messages

Exercising the TEST agent is quite limited. A better example is given by the Factorial example, taken from the list of test problems set up by the ASA group of the SMA College of AFIA (2003).

2.1.6 The Factorial Example

The factorial example is a well known example for testing multi-agent basic capabilities. The version described in this section was produced on the OMAS v 7.11 platform.

The factorial example normally contains only 4 agents:

- an agent called FACTORIAL that knows how to compute a factorial but does not know how to multiply two numbers;
- three multiplying agents (MUL-1, MUL-2, MUL-3) that know how to multiply two numbers, but do not how to compute a factorial.

In this section we show how various scenarios can be exercised.

In the factorial example agents share the same file. They could eventually share some code although agents use different packages. However, it is not the usual way to organize the agent files (as we shall see later), which explains that loading the file is slightly complicated. The interesting point here is that one can open a single file and look at the corresponding code easily.

To load the factorial demo, launch the lisp environment and activate the THOR-NULL application. An empty environment is activated. Then use the lisp "File" menu, clicking the "compile and load..." button to load the factorial.lisp file from the sample-applications folder of the OMAS folder. This loads the demo file and starts the four service agents FAC, MUL-1, MUL-2, MUL-3, that appear as SA_FAC, SA_MUL-1, SA_MUL-2 and SA_MUL-3 in the control panel. At the same time several predefined messages are loaded: DF, DFWTL, GF, CNF. Clicking on the "agents/msg" button displays the agents (Fig.3.1), then the messages (Fig.3.8).

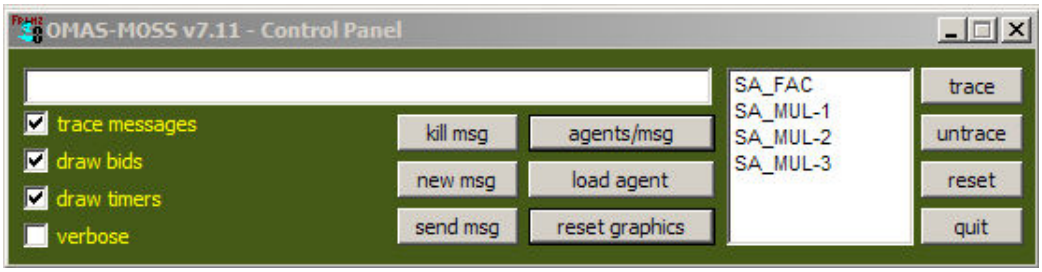

Figure 2.12: Agents from the factorial application

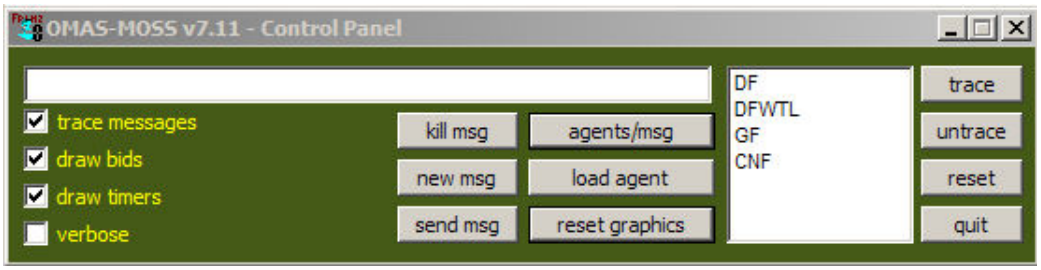

Figure 2.13: Messages for the factorial application

Dumb Strategy

The dumb strategy consists in subcontracting multiplies always to the same agent. It is implemented by the DF message and yields the result shown Fig.2.14.

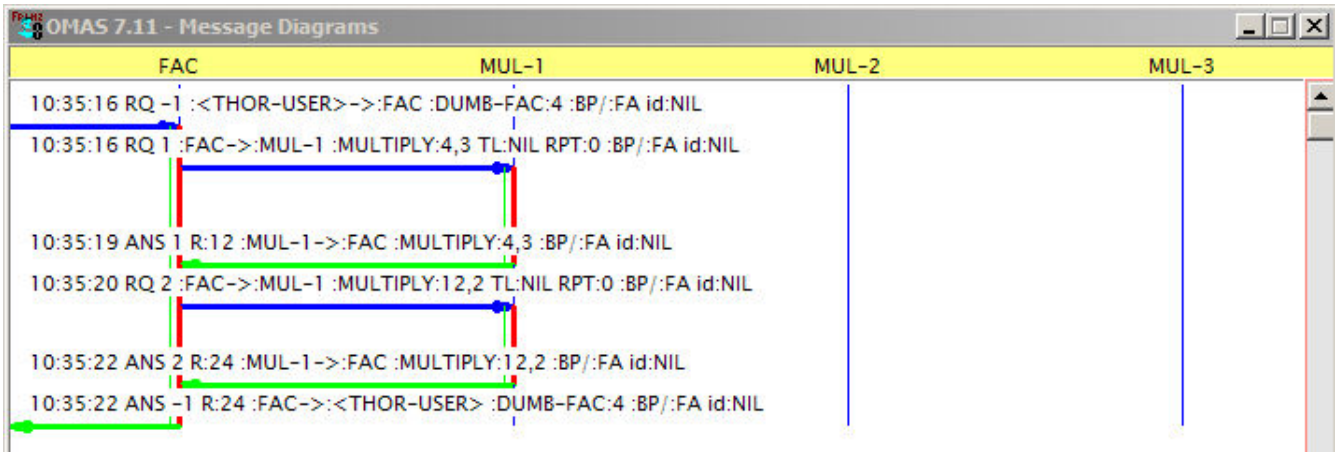

Figure 2.14: Computing 4! using the dumb strategy

The result, 24, can be seen in the text of the last message and appears in the answer line of the control panel. Note that there is a random waiting time to delay the answer of the MULTIPLY agents.

The skill of a MULTIPLY agent is rather straightforward. The agent receives 2 numbers, waits some time and returns the product.

```
(defskill :multiply :mul-1
  :static-fcn static-multiply
```

```
  )

(defun static-multiply (agent message n1 n2)
  (declare (ignore message))
  ;; slow down the process so that we can follow on the trace
  (sleep (1+ (random 3)))
  (static-exit agent (* n1 n2)))
```

In order to implement the dumb strategy, the FAC agent has a more complex skill than the TEST agent. Indeed, it starts by subcontracting some multiplies but must react when the answer comes back. Thus, its skill is divided into a `static` and a `dynamic` part. The static part launches the computation and the dynamic part is called whenever a result comes back from the multiply agents.

The code is the following:

```
;;;============================================================ DUMB STRATEGY
;;; using always MUL-1 to multiply

(omas::defskill :dumb-fac :FAC
  :static-fcn static-dumb-fac
  :dynamic-fcn dynamic-dumb-fac
  )

(defun static-dumb-fac (agent message nn)
  "Sends first multiply and set up ennvironment to keep track of where we are."
  (declare (ignore message))
  ;; if nn is less than or equal to 1, then return 1 immediately
  (if (< nn 2) (static-exit agent 1)
      ;; otherwise, create a subtask for computing the product of the first
      ;; two top values
      (progn
        ;; ship subtask to agent MUL-1 to compute
        (send-subtask agent :to :MUL-1 :action :multiply
                      :args (list nn (1- nn)))
        ;; define a tag (:n) in the environment to record the value of the next
        ;; products to compute. e.g., nn-2 -> (nn - 2)!
        ;; use the special environment area of the agent (associated with the task)
        (env-set agent (- nn 2) :n)
        ;; quit
        (static-exit agent :done))))

(defun dynamic-dumb-fac (agent message answer)
  "this function is called whenever we get a result from a subtask. This ~
      approach is not particularly clever, since the computation is linear ~
      and uses the same multiplying agent, i.e., MUL-1."
  (declare (ignore message))
  (let ((nn (env-get agent :n)))
    ;; if the recorded value is 1 or less, then we are through
    (if (< nn 2)
      ;; thus we do a final exit (in fact we should never go there ??
      (dynamic-exit agent answer)
      ;; otherwise we multiply the answer with the next high number
```

```
;; creating a subtask
(progn
  (send-subtask agent :to :MUL-1 :action :multiply
                :args (list answer nn))
  ;; update environment
  (env-set agent (1- nn) :n)
  ;; then the function returns a value to nobody in particular
  answer
  ))))
```

The static function sends a subtask (send-subtask) to MUL-1.

The dynamic part of the factorial receives an answer and must decide whether the computation is finished or not. When the computation is finished it calls the dynamic-exit function. Otherwise, it sends another multiply to MUL-1.

Intermediate results are saved into the environment of the task. The environment is structured as a property/value list. Here, the property is specified by the user-defined `:n` keyword.

`send-subtask`, `static-exit`, `dynamic-exit`, `update-environment` are functions from the OMAS API library.

Dumb Strategy with Time Limit

Here we set a time limit value on the computation of factorial. Since MUL-1 takes its time to answer, if we set the time limit to 5 seconds, then we'll observe an error (Fig.2.15). The time limit is set as a parameter of the message sent to the agent FAC.

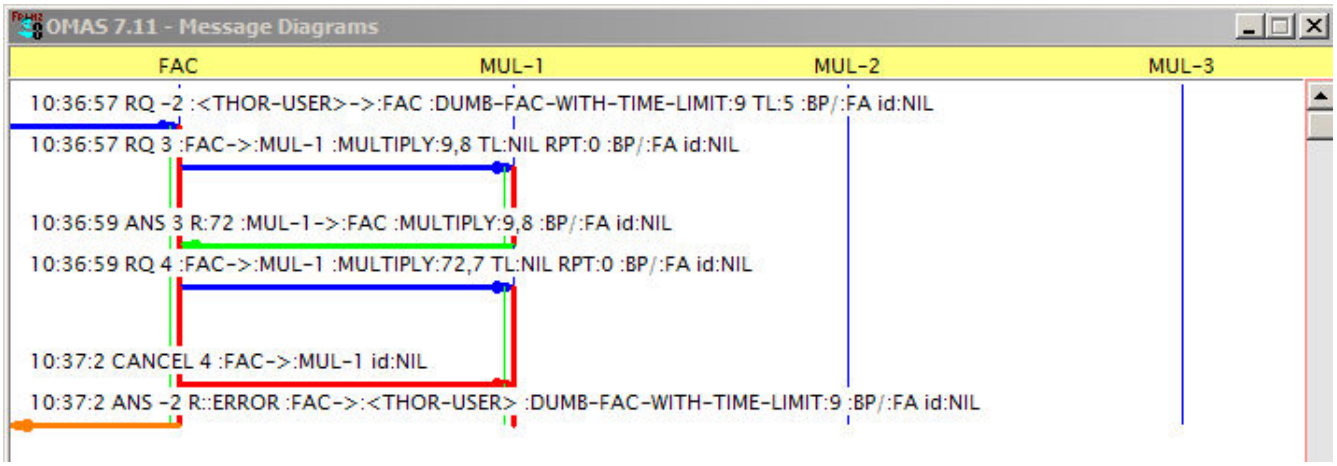

Figure 2.15: Computing 9! with a time limit of 5 seconds

In that specific case a time-limit handler is declared for the FACT agent as follows:

```
(omas::defskill :dumb-fac-with-time-limit :FAC
  :static-fcn static-dumb-fac
  :dynamic-fcn dynamic-dumb-fac
  :time-limit-fcn time-limit-dumb-fac
  )
with the corresponding code:

(defun time-limit-dumb-fac (agent dummy message)
```

```
  "when reaching a time-limit for computing the function we simply abort all ~
      subtasks and quit."
  (declare (ignore dummy message))
  ;; cancel current subtasks
  (cancel-all-subtasks agent)
  ;; and return with :error flag
  (dynamic-exit agent :error))
```

The triggering message, DFWTL, is defined as::

```
(defmessage :DFWTL :to :FAC :type :request
  :action :dumb-fac-with-time-limit :args (9) :time-limit 5)
```

Greedy Strategy

The greedy strategy distributes the computations among the various agents in turn as soon as an answer is returned. The result is shown Fig.2.16 for the computation of fac(9).

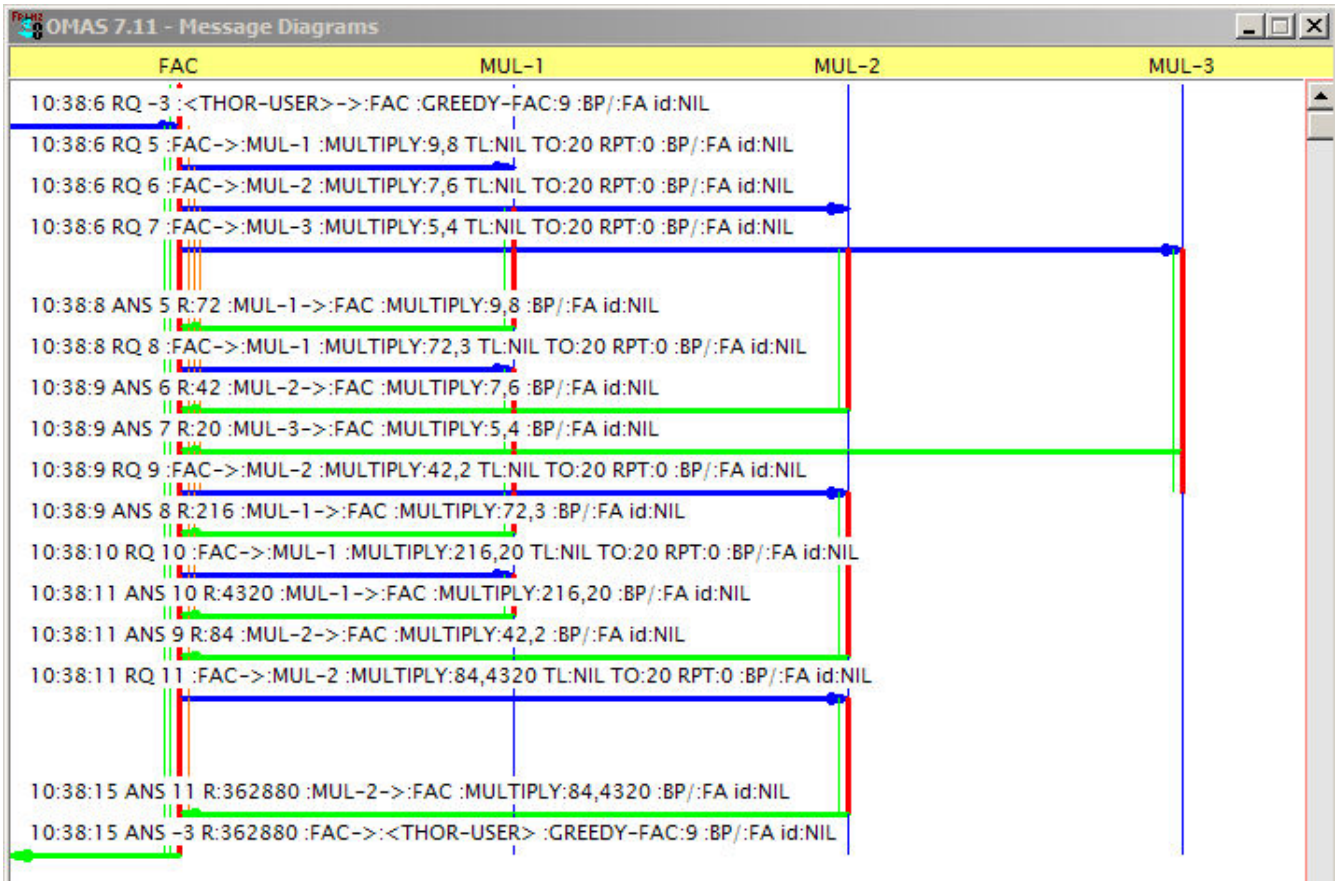

Figure 2.16: Computing 9! using the Greedy strategy

The code for the skills is the following:

```
(defskill greedy-factorial :FAC
  :static-fcn static-greedy-factorial
  :dynamic-fcn dynamic-greedy-factorial)
```

```
(defParameter *multiply-timeout* 20 "should be no problem with 20 seconds")

(defun static-greedy-fac (agent message nn)
  "Here we assume that we have 3 acquaintances that know how to multiply and ~
      we try to use them as best as we can. I.e., we distribute the first three ~
      multiplications, then each time an answer comes back we send a new multiplication ~
      to the agent that perfomed the subtask."
  (declare (ignore message)(special *multiply-timeout*))
  ;; if nn is less than or equal to 1, then return 1 immediately
  (if (< nn 2) (static-exit agent 1)
    ;; otherwise, create a series of subtasks for computing the product of the
    ;; first top values
    (let ()
      ;; we initialize a :res value to 1 in the environment (used by dynamic function)
      (env-set agent 1 :res)
      ;; ship subtask to agent MUL-1 to compute
      (send-subtask agent :to :MUL-1 :action :multiply
                    :args (list nn (1- nn)) :timeout *multiply-timeout*)
      ;; decrease nn
      (decf nn 2)
      ;; when nn is greater or equal than 2 continue
      (when (> nn 3)
        (send-subtask agent :to :MUL-2 :action :multiply
                      :args (list nn (1- nn)) :timeout *multiply-timeout*)
        ;; decrease nn
        (decf nn 2))
      ;; when nn is greater or equal than 2 continue
      (when (> nn 3)
        (send-subtask agent :to :MUL-3 :action :multiply
                      :args (list nn (1- nn)) :timeout *multiply-timeout*)
        (decf nn 2))
      ;; define a tag (:n) in the environment to record the value of the next
      ;; products to compute
      (env-set agent nn :n)
      ;; quit
      (static-exit agent :done))))

(defun dynamic-greedy-fac (agent message answer)
  "this function is called whenever we get a result from a subtask. "
  (declare (special *multiply-timeout*))
  (let ((nn (env-get agent :n))
        (res (env-get agent :res))
        (list-of-subtasks (pending-subtasks? agent)))
    ;(format t "~&Dynamic: nn=~s - (pending-subtasks? agent) ~S"
    ;        (cadr (member :res environment)) (pending-subtasks? agent))
    ;; we are finished when this was the last substasks and we do not have
    ;; partial result waiting (res=1)
    (cond
     ((and (< nn 2) (null (cdr list-of-subtasks))(eql res 1))
```

```
 ;; thus we do a final exit, we just received the final answer
 (dynamic-exit agent answer))
;; if the recorded value is greater than 1, then we send a new subtask to the
;; agent that sent the answer
((> nn 1)
 (send-subtask agent :to (sending-agent message) :action :multiply
               :args (list answer nn) :timeout *multiply-timeout*)
 ;; update environment
 (env-set agent (1- nn) :n)
 ;; then return an answer (actually to nobody in particular)
 answer)
;; otherwise we exhausted primary values, we must gather results of MUL
;; agents and multiply them together
((eql 1 res)
 ;; if 1 then we store partial result for next time around
 (env-set agent answer :res)
 ;; then return an answer (actually to nobody in particular)
 answer)
;; otherwise we have a partial result in the environment
((not (eql 1 res))
 ;; use it to send a new multiply job
 (send-subtask agent :to (sending-agent message) :action :multiply
               :args (list answer (env-get agent :res))
               :timeout *multiply-timeout*)
 ;; reset environment
 (env-set agent 1 :res)
 ;; then return an answer (actually to nobody in particular)
 answer)
;; otherwise we are in trouble (something wrong with the algorithm)
(t (error "bad greedy fac algorithm")))))
```

The trace on Fig.2.16 shows a reallocation to the MUL-2 agent after a deadline on the MUL-3 agent that cannot multiply big numbers.

Note that the send-subtask arguments include a `:timeout` argument `*multiply-timeout*` meaning that the FAC agent will wait only a limited time for an answer from a multiply agent. The value is set to 20 seconds, but can be decreased to say 3 seconds by typing:

```
? (setq *multiply-timeout* 3)
```

Contract Net Strategy

This strategy uses a different approach. Indeed, it sends call for bids using the Contract Net protocol. The FAC agent sends a call-for-bids (light blue messages, Fig.2.17) and MULTIPLY agents answer the bids offering services (brown messages)[5]. The FAC agent here takes the first bid that comes back and grants the job to the agent. The process is repeated until all multiplies are done.

Fig.2.18 shows the same process but bids have been removed from the Message diagrams. Obtained by un-checking the "draw bids" box from the control panel.

The code for the skills is the following:

[5]Brown messages only appear in the message diagrams window if the corresponding flag is active. If they do not appear, click the "draw bids" radio button of the control panel and send the message again.

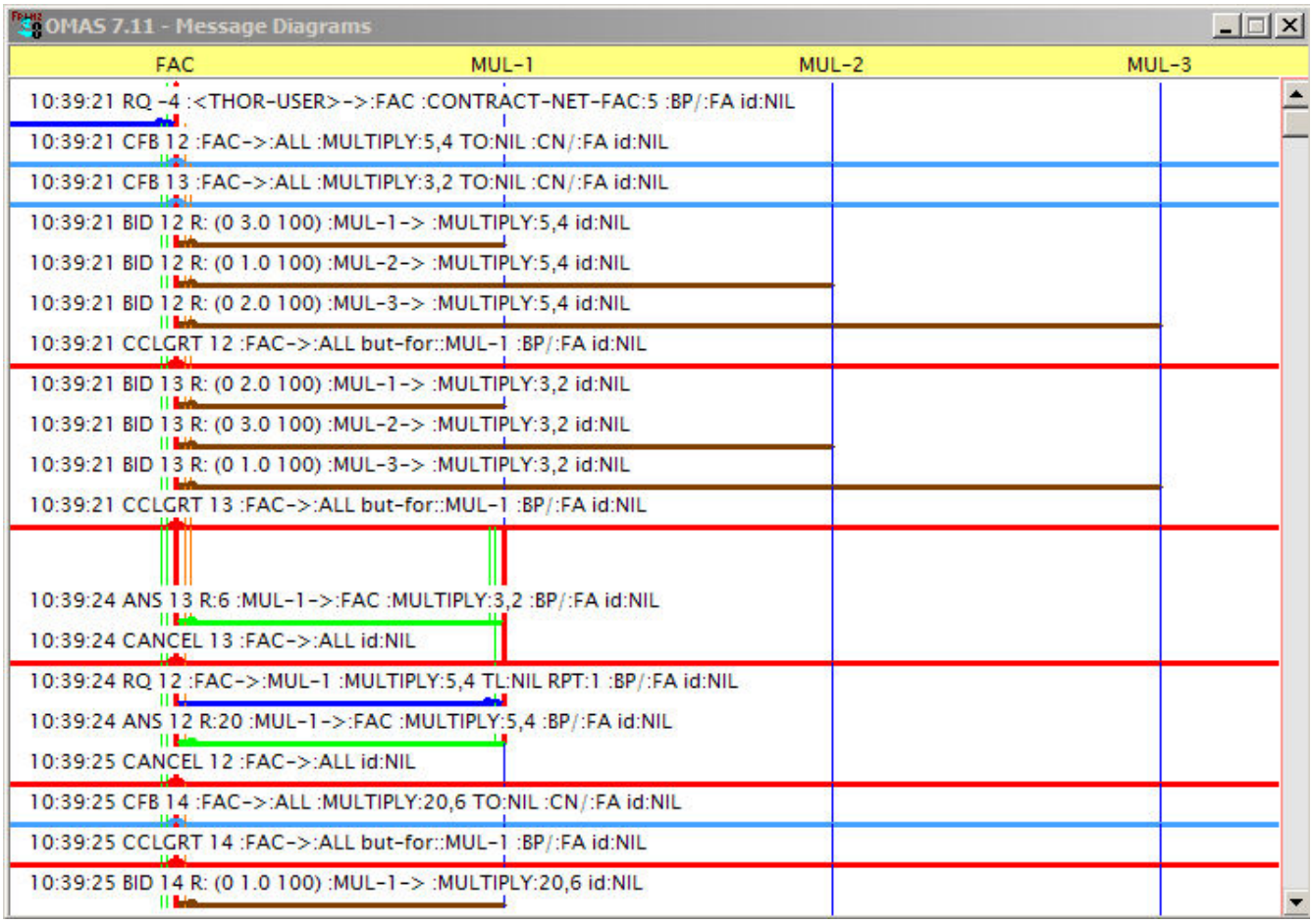

Figure 2.17: Computing 5! using a Contract-Net strategy

```
(defskill :contract-net-fac :fac
  :static-fcn static-contract-net-fac
  :dynamic-fcn dynamic-contract-net-fac
  )

(defun static-contract-net-fac (agent message nn)
  "this skill works by first creating n/2 tasks for distribution to the ~
      multiplying agents, i.e., (* n n-1) (* n-2 n-3) ...
  if n is even we end up with (* 2 1)
   if n is odd we end up with (° 3 2)
  It then broadcasts those tasks, record the left over number (1 or 2) and quits."
  (declare (ignore message))
  ;; we want to control the call for bid delay for the contract net to prevent early
  ;; aborts
  (if (< nn 2) (static-exit agent 1)
      ;; this line should be replaced with (if (< nn 3) (return (max nn 1))
      ;; to avoid subcontracting (* 2 1)
      (let ((delay 0))
        ;; try to produce as many initial subtasks as possible
        (loop
```

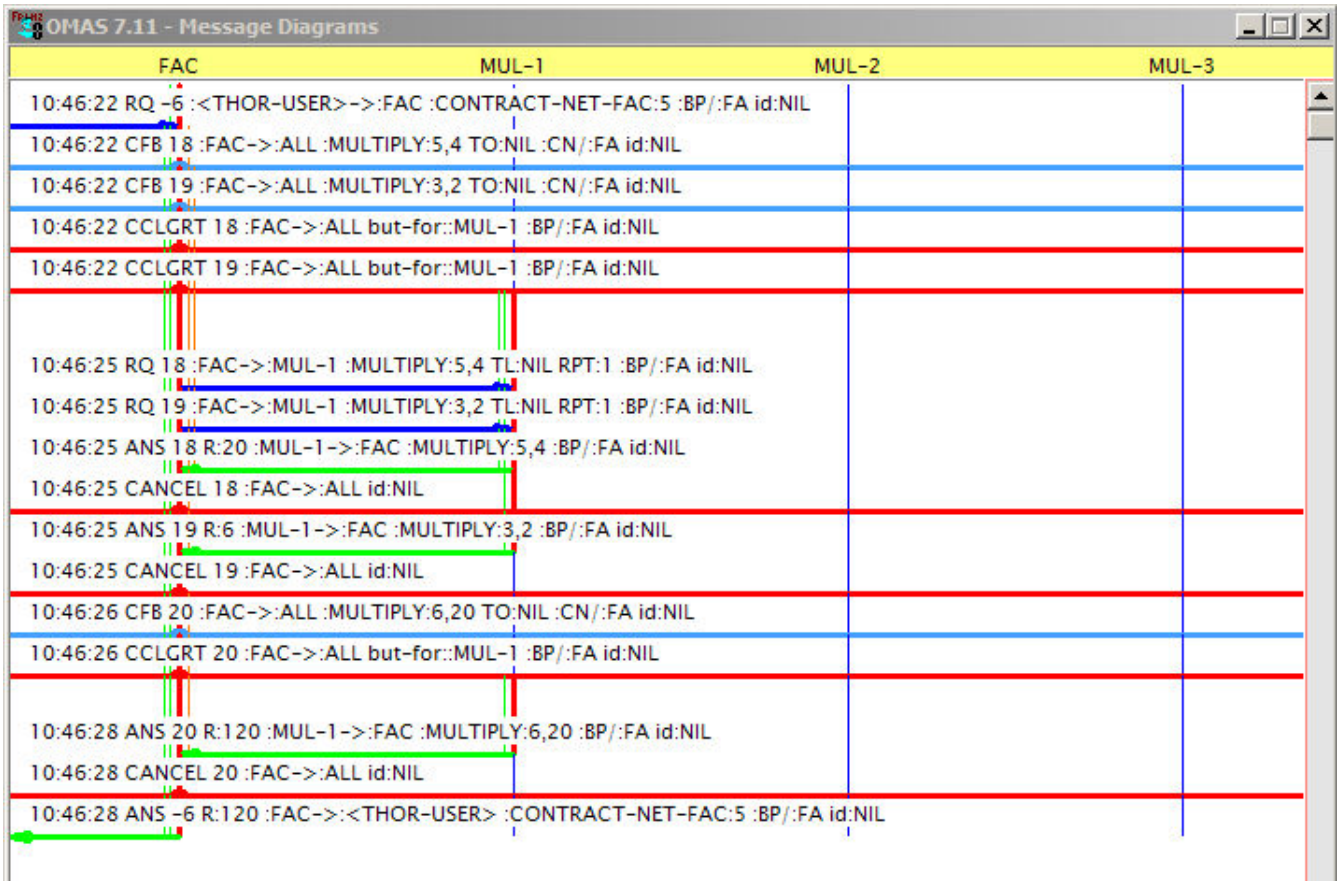

Figure 2.18: Computing 5! with a Contract-Net strategy hiding bids

```
          ;; create n/2 tasks for distribution to the multiplying agents
          ;; create a subtask for computing the product of the next two values
          ;; broadcast
          (send-subtask agent :to :ALL :action :multiply
                        :args (list nn (decf nn)) :delay delay
                        :protocol :contract-net)
          (decf nn) ; adjust nn for next round
          ;(incf omas::*default-call-for-bids-timeout-delay* 3) ; increase waiting time
          ;;... to let agents make their bids
          ;(incf delay) ; we assume 1 as cost of setting up subtask
          ;; if nn becomes less than 3 don't multiply 2 by 1! get out of the loop
          (if (< nn 3)(return nil)))
        ;; define the :n tag in the environment to record the value of the next
        ;; products to compute. When exiting the loop it can be 2, 1, or 0. We set
        ;; it to 2 or 1
        (env-set agent (if (< nn 2) 1 nn) :res)
        ;; quit
        (static-exit agent :done))))

(defun dynamic-contract-net-fac (agent message answer)
  "this function is called whenever we get a result from a subtask.
```

```
 We structured the environment as follow:
   :res contains a partial result (nil or 2 at the beginning)
 We use :res as follows:
   if 1 we save the result into it
   otherwise, we broadcast the multiplication of :res with the result."
(declare (ignore message))
(let (res)
  ;; we are finished when there are no more active substasks.
  (cond
   ((not (pending-subtasks? agent))
    ;; thus we do a final exit
    (dynamic-exit agent answer))
   ((eql 1 (setq res (env-get agent :res)))
    ;; we have still subtasks in progress and res is nil
    ;; store the answer
    (env-set agent answer :res)
    ;; check if this was the last subtask in the list
    (if (null (cdr (pending-subtasks? agent)))
      (return-from dynamic-contract-net-fac (dynamic-exit agent answer)))
    ;; otherwise return answer
    answer)
   (t
    ;; res is not 1, i.e., it contains a partial result
    ;; we multiply the answer with res creating a subtask
    ;; ... broadcasting it
    (send-subtask agent :to :ALL :action :multiply
                  :args (list answer res) :protocol :contract-net)
    ;; update environment
    (env-set agent 1 :res)
    answer))))
```

Notice that the main difference in the way of sending subtasks is to broadcast them (`:to ALL`) and to declare a Contract Net protocol (`:protocol :contract-net`). The actual Contract Net mechanism is taken care of by the OMAS platform.

2.1.7 Setting Up Your Own Application

Applications are constructed by specifying three things: (i) the agents, (ii) the skills, and (iii) the messages. We examine those in turn taking as examples the contents of the `factorial.lisp` file containing the FAC demo.

Defining Agents

Agents are defined simply, using the `defagent` macro. E.g.,

```
(defagent :fac)
(defagent :mul-1)
```

Defining Skills

Defining skills is done with the `defskill` macro. E.g.,

```
(defskill :MULTIPLY :MUL-1
     :static-fcn static-multiply)

(defskill :DUMB-FACTORIAL :FAC
     :static-fcn static-dumb-factorial
     :dynamic-fcn dynamic-dumb-factorial)
```

where `static-multiply`, `static-dumb-factorial`, `dynamic-dumb-factorial` represent Lisp functions implementing the skill as shown previously.

Defining Messages

Messages can be predefined as follows:

```
(defmessage :DF :to :FAC :type :request :action :dumb-fac :args (4))
```

Alternatively, they can be constructed interactively as needed, using the message edit window.

Running the Application

Actually, agents become active as soon as the `defagent` macros are executed. Then one can send them messages, using the control panel. OMAS agents when created stay alive and wait for something to happen. They have skills, i.e. capabilities of doing some tasks, different from goals (OMAS agents can also have goals). Thus, unless an agent is removed manually, or the machine on which it runs crashes, it stays alive ready to take part in the action. Note also that there is no White Pages nor Yellow Pages services.

File Organization

In this section applications were programmed in the same text file. In practice agents may be developed in the same Lisp environment or dispatched on different machines. Therefore, each agent will have its own set of files and be developed in its own name space to avoid unintended interactions.

How to develop complex applications is presented in Section 2.2.

2.1.8 Distributing Agents

Distributing agents is more complex since one has to define a set of files for each agent to be able to install them on different machine. See Section 2.2.7.

2.2 Tutorial 2: Building an Application

This tutorial shows how to create OMAS agents and how to make them execute tasks and communicate with one another. OMAS is completely written in Lisp and OMAS programmers work in Lisp when developing their agents. Therefore, the reader is assumed to be familiar with the Lisp programming language.

2.2.1 OMAS Overview

OMAS is an environment that facilitates the development of cognitive multi-agent systems. It includes

- a **runtime environment** where OMAS agents can "live" and that must be active on a given host before one or more agents can be executed on that host;
- a set of **prototyped agents** that programmers can select to develop their applications;
- a suite of **debugging tools** that allow debugging and monitoring the activities of running agents.

Coteries and Platform

Each running instance of the OMAS runtime environment is called a *local coterie* and may contain several agents. The set of local coteries is called a *coterie*. One or more coteries is called a *platform*. There is no hierarchy among the coteries. A coterie is physically local, i.e. it must reside on the same loop of a LAN. All agents are connected on the same port.

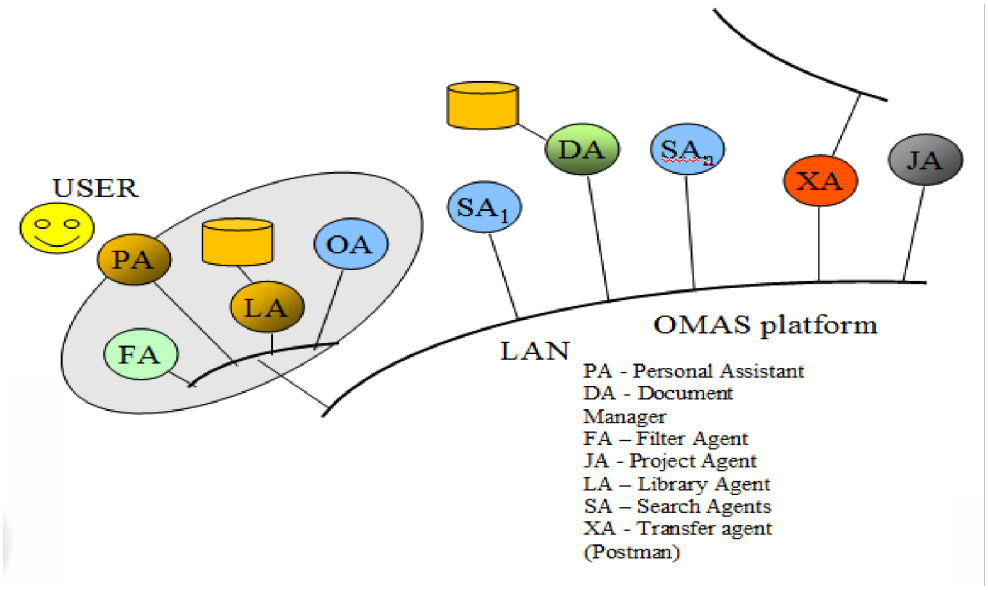

Figure 2.19: Example of OMAS coterie

Coteries not located on the same LAN loop may be connected by a special agent called a transfer agent or *postman* that acts as a gateway. Inside a platform agents must have unique names.

Fig.2.19 shows an example of coterie. Agents in the oval section are located in the same Lisp environment on the user's machine (advised but not required), other agents are distributed on the network and may be on the same machine or on different machines, the XA agent connects this coterie with possibly other coteries or external legacy systems.

An OMAS environment is started by loading the OMAS application inside a Lisp environment (ACL on Windows or MCL on Macs).

Agents Organization

Because agents of the OMAS platform communicate using broadcast messages, registries like white pages or yellow pages are not necessary.

2.2.2 Developing an Application

Developing a multi-agent application is a difficult endeavor. One must first define the global architecture of the targeted system, determine how many agents will be needed, and which skills such agents must possess. Analysis and design methods are available to do that. A popular method is Gaia as proposed by Wooldridge [**?**]. Another one is SAAS as developed by De Azevedo [**?**].

Once the global architecture of the MAS is determined, one needs to actually program the agents, then test the results. To this effect a platform will provide the necessary tools and usually impose the programming language. OMAS uses Lisp, although Java can be used through Lisp-to-Java links.

MAS applications can be ranked according to their level of difficulty. One can distinguish:

- **level 0:** involving only service agents, e.g. for building an automatic application on a LAN loop;
- **level 1:** involving several local distant coteries and requiring transfer agents or postmen;
- **level 2:** involving agents and outside applications (e.g. legacy software);
- **level 3:** involving interaction with a user to drive the application;
- **level 4:** involving several persons with their own personal assistants possibly using different natural languages.

Sections 2.2.3 to 2.2.5 develop a small example illustrating the use of a service agent and of a personal assistant agent.

2.2.3 The "Calendar" Example

This section introduces a simple example consisting of a single service agent named CALENDAR and a French speaking Personal Assistant agent named OSCAR. We want the CALENDAR to give us the current date and will later develop OSCAR as the interface with the system.

In order to do so we first open the OMAS/applications folder and create a copy of the application-templates folder that we rename UTC-TEST to host our TEST application, leaving it in the OMAS/applications folder.

Our UTC-TEST folder contains several files (copied from the application-templates folder):

- agents.lisp
- SAAA.lisp
- SAAA-ONTOLOGY.lisp
- PBBB.lisp
- PBBB-ONTOLOGY.lisp
- PBBB-TASKS.lisp
- PBBB-DIALOG.lisp

- XCCC.lisp
- XCCC-ONTOLOGY.lisp
- IA-DDD.lisp
- Z-MESSAGES.lisp

Each file is a template for developing necessary files for the application:

- SAAA and optionally SAAA-ONTOLOGY are used to build a Service agent
- PBBB, PBBB-DIALOG, PBBB-ONTOLOGY, and PBBB-TASKS are used to build a personal assistant agent
- XCCC and optionally XCCC-ONTOLOGY are used to build a transfer agent of postman
- IA-DDD is used to build an Inferer agent (set of rules)
- Z-MESSAGES is used to predefine test messages
- agents is used to load the local coterie, namely all files that have been defined

2.2.4 Creating the CALENDAR Service Agent

Our CALENDAR agent is simple, we want it to return the date when we send it the message :get-date, as a string containing the day, month and year and eventually the current time. It will thus have a single skill :GET-DATE and return a string. We do not need an ontology to do that.

The CALENDAR File

To create the CALENDAR file:

1. Rename the file SAAA.lisp CALENDAR.lisp
2. Open the file with an editor supporting UTF-8[6], e.g. Notepad++ or Emacs. We found that the ACL 8.2 editor introduces spurious characters in UTF-8 encoded files. However, the ACL 9.0 Lisp editor is fine.
3. Replace SAAA by CALENDAR throughout the file.
4. Go to the Skill section.

Note that the file contains three lines of Lisp code for creating a specific package for hosting the agent, setting this package to be the current package for the file, and for defining the agent.

```
...
;;;============================================================================
;;;
;;;                          defining agent package
;;;
;;;============================================================================

(defpackage :CALENDAR (:use :moss :omas :cl))
(in-package :CALENDAR)
```

[6]All standard agent files use a UTF-8 encoding for supporting different languages.

```
;;;============================================================================
;;;
;;;                              defining the agent
;;;
;;;============================================================================
;;; :CALENDAR is a keyword that will refer to the agent and be used in the messages.
;;; The actual name of the agent is built automatically and is the symbol
;;; CALENDAR::SA_CALENDAR, i.e. the symbol SA_CALENDAR defined in the "CALENDAR" package.
;;; It points to the lisp structure containing the agent data

(omas::defagent :CALENDAR)
...
```

The GET-DATA Skill

The skill section is included in comments as follows:

```
;;; ============================================================================
;;;                                 Skill  XXX
;;; ============================================================================
#|
(defskill :XXX :SAAA
  :static-fcn static-XXX
  :dynamic-fcn dynamic-XXX
  :acknowledge-fcn acknowledge-XXX
  :bid-cost-fcn bid-cost-XXX
  :bid-quality-fcn bid-quality-XXX
  :bid-start-time-fcn bid-start-time-XXX
  :how-long-fcn how-long-XXX
  :preconditions preconditions-XXX
  :select-best-answer-fcn select-best-answer-XXX
  :time-limit-fcn time-limit-XXX
  :timeout-handler timeout-handler-XXX
  )

;;; Keep only those functions that are useful for your application
;;; agent and message are variables set by OMAS
;;; agent is the current agent
;;; message is the message that trigerred the function

(defun static-xxx (agent message <args>)
  "documentation"
  (static-exit agent <result>))

(defun dynamic-xxx (agent message <args>)
  "documentation"
  (dynamic-exit agent <result>))

(defun acknnowledge-xxx (agent message <args>)
    "documentation"
```

```
  )

;;; <arg-list> is the list of args contained in the call-for-bids message, i.e.
;;; the args corresponding to the task to be done

(defun bid-cost-fcn bid-cost-xxx (agent <arg-list>)
    "documentation"
  <return a list (<cost> <rigidity>)>)

(defun bid-quality-xxx (agent <arg-list>)
    "documentation"
  )

(defun bid-start-time-xxx (agent <arg-list>)
    "documentation"
  )

(defun how-long-xxx (agent <arg-list>)
    "documentation"
  )

(defun preconditions-xxx (agent message <args>)
    "documentation"
  )

(defun select-best-answer-xxx (agent answer-message-list)
    "documentation"
  )

(defun time-limit-xxx (agent message message)
    "documentation"
  )

(defun timeout-handler-xxx (agent message)
    "documentation"
  )
|#
```

This part of the file contains all possible options for a particular skill. Since we want a simple skill with no subtask transferred to other agents, we only need the static part of the skill.

The date is obtained from the standard decode-universal-time and get-universal-time Lisp primitives, which leads to the following code for the skill:

```
;;; ============================================================================
;;;                                  Skill  GET-DATE
;;; ============================================================================

(defskill :GET-DATE :CALENDAR
  :static-fcn static-GET-DATE
  )
```

```
;;; Keep only those functions that are useful for your application
;;; agent and message are variables set by OMAS
;;; agent is the current agent
;;; message is the message that trigerred the function

(defUn static-get-date (agent message)
  "documentation"
  (let (hour minute second)
    (multiple-value-setq (second minute hour day month year)
      (decode-universal-time (get-universal-time)))
    (static-exit agent
         (format nil "~S/~S/~S ~S:~S:~S"
                         day month year hour minute second))))
```

Save the edited CALENDAR.lisp file.

Loading the Application

In order for the CALENDAR file to be loaded as an application, we need to declare the agent name in the agents.lisp file replacing the default values as follows:

```
...
;;; replace the keywords :SAAA-1, :SAAA-2, :PBBB, :XCCC with the keywords specifying
;;; your agents. They will name the agent files. Remove the three dots ...

(setq *local-coterie-agents*
      '(:CALENDAR))

:EOF
```

Note that the agent is declared as a keyword :CALENDAR in the list of agents.

Now, after loading the OMAS platform one will specify the application in the OMAS menu (Fig.2.20). The name of our folder, UTC-TEST, has two parts, the first one, UTC, is a reference to the local coterie (also called site), the second part, TEST, is the name of the application.

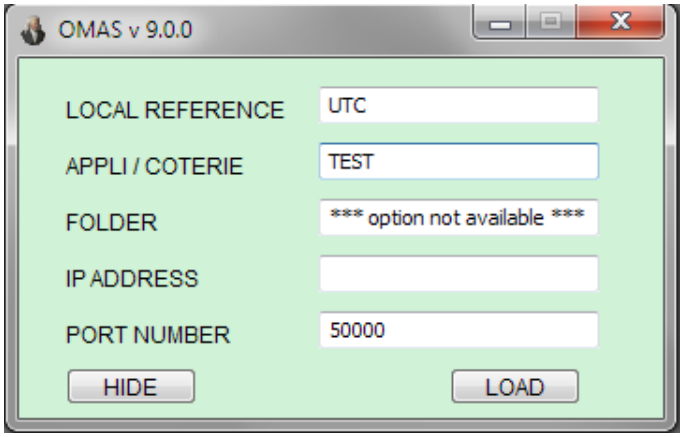

Figure 2.20: OMAS Initial Menu

When we start the application by clicking LOAD, the content of the initial window will be saved for the next session. Once the application is launched we can send a new request message to CALENDAR with a :get-date action as shown in the first tutorial of this chapter. The result will appear in the top line of the control panel.

Note on Fig.2.20 that the IP ADDRESS field is left blank, meaning that we do not send messages on the network for the time being (thus, port 50000 is unused).

2.2.5 Creating the OSCAR Personal Assistant Agent

Creating OSCAR is done by using the PBBB files. As a first step we'll use the PBBB file and the PBBB-DIALOG file that provides a minimal dialog.

Creating the OSCAR Files

To create the OSCAR files do the following:

1. Rename the PBBB.lisp file to OSCAR.lisp
2. Open the file and replace PBBB by OSCAR everywhere
3. Rename the PBBB-DIALOG.lisp file to OASCAR-DIALOG.lisp
4. Open the file and replace PBBB by OSCAR everywhere
5. Close and save the files

2.2.6 Adding OSCAR to the Application

To do so:

1. Open the agent.lisp file
2. Add OSCAR to the list of agents

```
(setq *local-coterie-agents*
      '(:CALENDAR :OSCAR))
```

3. Close the file, reinitialize the Lisp environment, reload OMAS and reload the application without changing the OMAS initial menu.

A personal assistant default window should appear (Fig.2.21).

Now, OSCAR cannot do much although it has some initial dialog. OSCAR can answer "Bonjour." or "Bonjour OSCAR." but otherwise cannot perform any task.

Letting OSCAR Do Some Tasks

In order for OSCAR to be able to do some tasks, we must define some tasks and provide the corresponding dialogs for activating such tasks. To do so, one must provide a specific file obtained by renaming the PBBB-TASKS.lisp to OSCAR-TASKS.lisp. Like in the previous steps, open the file and replace PBBB by OSCAR everywhere and reload everything.

Now the agent has four predefined tasks:

- HELP that lists the possible tasks

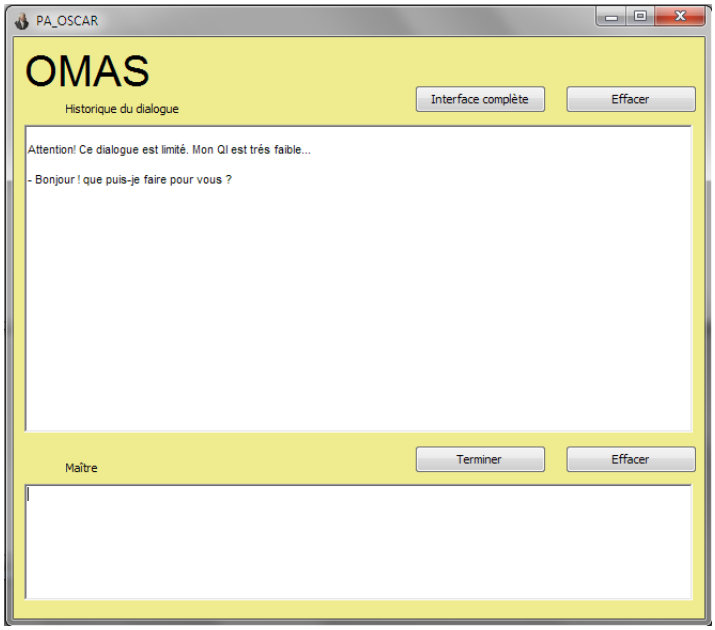

Figure 2.21: OMAS Initial Menu

- SET FONTS that changes the size of the font
- TRACE that traces dialog when debugging
- WHAT IS that tries to answer the question

However, this is not very helpful. You can try "Lettres plus gros." and see what happens. The WHAT IS task may not answer if no agent knows anything about the question.

Creating a New Task and Dialog

Now we would like to create a new dialog for asking OSCAR the current date. If we ask that, OSCAR will have to get the current date from the CALENDAR agent.

First we have to gather the possible expressions that people could use to ask for the current date. They may include phrases like:

- "Aujourd'hui c'est quel jour?"
- "Quel jour sommes-nous aujourd'hui ?"
- "Date de ce jour ?"
- "Aujourd'hui c'est quelle date ?"
- "Date ?"
- etc.

From such expressions we can extract cues that will trigger the GET-DATE task that will send a :get-date request to the CALENDAR agent. With such cues, we build a task that could be:

```
(deftask "get date"
   :doc (:en "Task to get the current date."
         :fr "Tâche permettant d'obtenir la date du jour.")
    :performative :request
    :dialog _get-date-conversation
    :indexes ("aujourd hui" .4 "jour" .3 "date" .6 "de ce jour" .4)
    )
```

The weights following the indexes are set manually and reflect the importance of a particular phrase in the input sentence with respect to this task.

Now the task refers to a _get-date-conversation that will be triggered if OSCAR determines that we are interested in a date. Save the OSCAR-TASKS.lisp file and close it.

The conversation corresponding to the _get-date-conversation task has to be added to the content of the OSCAR-DIALOG file, and could resemble the following code:

```
;;;============================================================================
;;;
;;;                        GET DATE CONVERSATION
;;;
;;;============================================================================

;;; this conversation is intended to obtain the date from the CALENDAR agent

;;;------------------------------------------------- (GD) GET-DATE-CONVERSATION

(defsubdialog
  _get-date-conversation
  (:label "Get Date conversation")
  (:explanation
   "We ask our PA the date of the day.")
   (:states _get-date-entry-state)
  )

;;;============================================================================
;;;                          GET DATE CONVERSATION STATES
;;;============================================================================

;;;------------------------------------------------- (GD) GET-DATE-ENTRY-STATE

(defstate
  _get-date-entry-state
  (:entry-state _get-date-conversation)
  (:label "Get date dialog entry")
  (:explanation "Assistant is sending a message to the CALENDAR agent.")
  (:send-message :to :CALENDAR :self *current-agent* :action :get-date)
  (:transitions
   (:always :display-answer :success))
  )
```

In this example the dialog has a single state (_get-date-entry-state) in which one sends a message to

the CALENDAR agent and prints the answer, then exiting with a success. The underscore in front of the symbols denote an OMAS variable.

Thus, add the code to the content of the OSCAR-DIALOG.lisp file, add :OSCAR to the list of agents in the agents.lisp file, reload the application, and try some things like: "Quel jour sommes-nous aujourd'hui?" or simply "Date?" and see what happens.

2.2.7 Distributing Agents

MAS applications are interesting when agents are distributed on different machines. In the case of the OMAS platform, agents cannot currently migrate from a machine to a different machine automatically. However, they can be easily installed on different machines. We distinguish two cases: (i) local coteries; and (ii) coteries with distant local coteries.

Local Coteries

A local coterie is a set of agents residing on the same physical LAN loop. This is an interesting configuration because one can use a single UDP message to implement broadcasts. In our example we can migrate the Calendar agent to a different machine, say UTC@SKOPELOS by creating a new OMAS environment on this machine and installing the CALENDAR files in the applications folder of this new environment. We need to remove the CALENDAR agent from the list of local agents in the first machine (by editing the agents.lisp file), and set the local agent list on SKOPELOS to contain OSCAR.

Finally one must decide which port to use on the local network and provide the broadcast address to the applications by filling the IP ADDRESS of the initial menu. For example, if we are on a 172.18 loop, we can set the IP ADDRESS to 172.18.255.255 and use the default PORT NUMBER of 50000.

There is nothing more to do.

Coteries Composed of Distant Local Coteries

When distributing agents among distinct local coteries, the situation is different because we must provide a communication channel between the two local coteries. This is done by using a transfer agent or postman. If we have OSCAR on local coterie LCA and CALENDAR on local coterie LCB, we must install a postman to LCB in LCA and a postman to LCA in LCB. LCA and LCB may be physically located anywhere in the world as long as firewalls do not block access to the communication port (more on that later).

2.3 Tutorial 3: Advanced Features

This third tutorial introduces some of the features and options that can be used for creating more complex applications. They are introduced on examples. Details of the different features will be described in the following chapters (in particular in Chapter 5). Features are introduced roughly in order of increasing complexity.

2.3.1 IDE Features

How to use the IDE is fully described in Chapter 3.

2.3.2 Ignoring Requests

OMAS agents are not required to answer messages, meaning that they can ignore messages even if they have the skills to process them.

For example if a MULTIPLY agent does not like to multiply even numbers, then it can simply ignore messages containing even numbers. The behavior is implemented by returning the keyword :abort at the skill level.

```
(defagent :mul-4)

(defskill :multiply :mul-4
   :static-fcn static-multiply-4)

(defun static-multiply-4 (agent message n1 2)
   (declare (ignore message))
   (static-exit agent (if (and (evenp n1)(evenp n2)) :abort (* n1 n2))))
```

One can check using the IDE that the agent indeed does not answer when the two numbers are even.

2.3.3 Checking Arguments

Sometimes it is necessary to dynamically check the arguments of a message before deciding to consider it or not. This can be done with the `:preconditions` option of the `defskill` macro.

Definition Preconditions implement a test on the arguments associated with a skill.

Mechanism When a preconditions function has been defined, it is executed prior to taking into account the corresponding subtask. If it returns nil, then the subtask is disregarded.

Specifying a Preconditions Function

```
(defskill :MULTIPLY MUL-1
  :static-fcn multiply
  :preconditions-fcn preconditions-multiply
  )

(defun preconditions-multiply (agent environment n1 n2)
  "called before triggering the skill?
  (declare (ignore agent environment))
  (and (integerp n1) (integerp n2)))
```

Preconditions are not restricted to checking the type of the arguments. One could imagine that the multiplying agent does not know how to multiply large numbers for example. Thus, the precondition will fail if one of the numbers is too large.

2.3.4 Specifying a Timeout

In OMAS we adopt the approach that if agents are allowed not to answer, then one must be able to specify timeouts easily. There may be different reasons for a request not to be answered: either agents are dead, or no agent wants to answer. In both cases something must be done at the application level.

Specifying a timeout can be done in two different ways:

1. by filling the timeout slot in the message edit window before sending the message;
2. by adding a :timeout argument in the send-subtask call within a skill.

```
(send-subtask agent :to (answering-agent agent) :action :multiply
                    :args (list answer (cadr (member :res environment)))
                    :timeout 2.5)
```

Timeouts are given in seconds, e.g. 2.5 means two and a half seconds. Note that timeouts concern the agent that sends the message.

Default behavior The subtask will be aborted after 2.5 seconds if no timeout handler has been defined (default behavior). Then, by default the task itself will be aborted.

Providing a Timeout Handler Specifying a user-defined timeout handler can be done by using the `:timeout-handler` option of the `defskill` macro. E.g.

```
(defskill :fast-fac-with-timeout fac
  :static-fcn static-fast-fac-with-timeout
  :dynamic-fcn dynamic-fast-fac-with-timeout
  :timeout-handler timeout-fast-fac-with-timeout)

(defun timeout-fast-fac-with-timeout (agent message)
  "function for handling timeout errors"
  (case (omas::action message)
    (:multiply
     ;; if a multiply subtask is timed out, then reallocate to another agent
     ;; by simply doing another broadcast
     (send-subtask agent :to :ALL :action :multiply
                   :args (omas::args message)
                   :timeout *multiply-timeout*)
     ;; exit
     :done
     )
    (otherwise
     ;; for any other skill we let the system process the timeout condition
     :unprocessed))
  )
```

On timeout, the handler is called and run.

The handler arguments are the agent structure and the message that was sent (containing the :timeout option). Parts of the message can be accessed by using (yet undocumented) internal OMAS functions. In the example `omas::action` retrieves the action specification of the message, and if the action is `:multiply` executes a new broadcast.

Note An important point is that there exists a single timeout-handler within a given task, which means that it could be used for different types of subtasks.

Another important point is that the handler is usually not run in the agent package. Thus, if agent information must be accesses the package must be specified by setting the special *package* variable temporarily, e.g.

```
(let ((*package* (find-package :oscar))) ...)
```

In practice, a timeout constraint is implemented through a timer set up in the agent sending the message.

2.3.5 Specifying a Time Limit

A time limit is the maximum time allowed to an agent for executing a task.

A time-limit can be specified in a message by using the `:time-limit` option, e.g. specifying a time-limit of 20 seconds for a subtask.

```
(send-subtask agent :to (answering-agent agent) :action :multiply
                    :args (list answer (cadr (member :res environment)))
                    :time-limit 20)
```

Default Behavior By default when a time limit is reached OMAS gives two more possibilities (extending the time-limit twice). On the third retry, the task is aborted.

Specifying a Time Limit Handler This is done by using the `:time-limit-fcn` option of the `defskill` macro.

```
(defskill :fast-fac fac
  :static-fcn static-fast-fac
  :dynamic-fcn dynamic-fast-fac
  :time-limit-fcn time-limit-fast-fac)

(defun time-limit-fast-fac (agent environment message)
  "function for handling time-limit errors"
  (declare (ignore environment)
  (format t "~&Warning; we went over time to process subtask ~S"
            (omas::task-id message))
  (abort-current-task agent))
```

Normally, the user aborts the task in the handler. However, when this is not the case, the default behavior takes over, i.e., two retries are allowed.

Usage Of course a time limit option can be used to tell an agent that its time to execute a specific skill is limited. Thus, if the agent knows that it cannot execute the subtask within the specified time, it can avoid undertaking it. In practice this is difficult to evaluate. Hence the time limit mechanism is rather used by the system internally to avoid orphaned tasks. OMAS associates a time limit of 1 hour to each task. Thus, after 1 hour, the task is aborted, cleaning the agent environment.

Another possible use is one involving human intervention. When a subtask must be solved by a human, then the 1 hour deadline may be too short and it is possible to specify a longer time-limit for executing the subtask.

2.3.6 Using Broadcast

We use broadcast when we want to send the same message to all reachable agents. Three strategies are possible and it is possible to specify the type of broadcast that we want to have within each message, by using the `:strategy` option. The allowed values are `:take-first-answer` (default), `:collect-answers` or `:collect-first-n-answers`.

The :take-first-answer Strategy

The `:take-first-answer` strategy is self explanatory. As soon as the sender gets an answer, it accepts it and calls the dynamic part of the skill.

The :collect-answers Strategy

The `:collect-answers` strategy is more complex, since in this case the sender waits some time (before a timeout occurs) collecting answers (actually answer messages). On timeout, a function is called to select the "best" answer from the collection of answers. If no function is given, then by default a message is picked randomly. A better approach consists in providing a selection function using the `:select-best-answer-function` of the `defskill` macro.

The following example, taken from the EDT application, illustrates how the function can be used to keep all the returned values and pass them to the dynamic part of the skill.

```
(defskill :GET-TIME-SLOT-CONSTRAINTS :RC
  ;; Before anything can be done, must obtain all constraints from the groups and
  ;; teachers, then set the status to :ready (the info is needed to draw the time
  ;; table). it is kept in RC's memory.
  ;; The collection is done by means of a broadcast, associated with a timeout
  ;; handler. All agents that have not answered prior to the timeout are considered
  ;;  not to have constraints.
  :static-fcn static-get-time-slot-constraints
  :dynamic-fcn dynamic-get-time-slot-constraints
  :select-best-answer-fcn best-answer-get-time-slot-constraints
  )

(defun best-answer-get-time-slot-constraints (agent message info)
  "called to select among the answer received from the broadcast.
Arguments:
   agent: RC
   environment: ignored
   info: list of answer messages
Return:
   list of contents of the answer messages."
```

```
(declare (ignore agent message))
;; transmit the list of all answers to the dynamic skill fcn
(mapcar #'omas::contents info))
```

The user handler accepts three arguments: the agent structure, the message just received, and the list of answer messages. In the example, the omas contents accessor extracts the content of the answer from each answer message. The list of returned values is then passed to the dynamic part of the skill.

Note 1: The handler function is executed in the context of the *timeout process*, the passed value will be used in the context of the *task process* (executing the skill).

Note 2: The second argument (message) of the collect user handler is not used and will be soon removed to let only two arguments: agent and list of messages.

The :collect-first-n-answers Strategy

Starting with OMAS version 7.10 a new strategy has been introduced. The rationale is the following: in some cases we send a broadcast that goes to all agents, although we know how many agents can answer. Thus, if all expected answers have been received, it is useless to wait until the broadcast timeout to process the answers. The :collect-first-n-answers has been designed to start processing as soon as all expected answers have been received. It is called by inserting a list into the send-subtask function:

```
:strategy '(:collect-first-n-answers 3)
```

2.3.7 Request for Acknowledgment

Sometimes it is useful to know if a message has reached some agent, in particlar when the answer might take a long time, or if one is not sure that the agent is still alive. For such cases, we can use the :ack option in a message.

Definition When an agent receives a message containing the `:ack` option, it then returns an acknowledge message before trying to process the received message. Sending an acknowledge message does not imply that the agent will process or answer the received message. It simply indicates that the message was received.

Example

```
(send-subtask :to :joe :action :confirm-appointment :args ("tuesday 10 am") :ack t)
```

When the acknowledge message returns we may want to process it independently from the task dynamic part. To do so we may define an acknowledge handler.

Specifying an Acknowledge Handler This is done using the `:acknowledge-fcn` option of the `defskill` macro.

```
(defskill :CONFIRM-APPOINTMENT TOM.
  :static-fcn static-confirm-appointment
  :dynamic-fcn dynamic-confirm-appointment
  :acknowledge-fcn acknowledge-confirm-appointment
  )
```

```
(defun acknowledge-confirm-appointment (agent message info)
  "called when receiving an acknowledge message?
  (declare (ignore message))
  ;; memorize whatever info was returned
  (remember agent info :appointment-request-acknowledgement))
```

The arguments to an acknowledge handler are the agent, the task message and the arguments of the subtask message.

2.3.8 Using Contract-Net

Contract-Net is a rather complex protocol in which an agent, the *manager*, outputs a call for bids (contract) and waits for answers in order to assign the job to one or more agents. Thus it is a 2-phase protocol.

Specifying a contract-net protocol is done on a per message basis as follows:

```
(send-subtask agent :to :ALL :action :multiply :args (list answer res)
          :protocol :contract-net)
```

By default the message will be transformed into a :call-for-bids and the manager will wait for answers until a timeout or until all answers have come back in a multicast.

Each bid is returned with a list of 5 elements: (start-time delay quality cost rigidity). Default start-time is 0 (meaning that the job can start right away), default duration is 3600 (1 hour, which is the default time limit for a task), default quality is 100 (maximum), default cost is 0, default rigidity is 100 (maximum).

Bids are then analyzed and ranked by cost first, completion time second (start-time + delay), and quality third. All best equal bids are selected. The task is then granted to all agents corresponding to best bids and the first answer coming back is accepted.

Several functions can be specified at the skill level to compute various parameters:

- bid-cost-fcn computes the cost (number)
- bid-quality-fcn returns an integer (0, 100)
- bid-start-time-fcn returns an integer (seconds)
- bid-how-long-fcn returns an estimate of the duration of the task (seconds)

If the comparison between two bids requires more sophisticated approaches, then the protocol must be programmed manually.

2.3.9 Specifying a Content Language

Normally, applications can decide on the content language for exchanging information within messages. Content languages range from logical languages to natural languages. OMAS does not impose any specific content language. However, when dealing with ontologies a simple syntax is proposed as mentioned in Section 9.2.

2.3.10 Setting up Goals

Agents are autonomous programs. As such they can have *goals*. OMAS allows defining one-shot or cyclic goals as described in Section 5.3.

2.3.11 Defining Dialogs

When interacting with a user, personal assistants use dialogs. Dialogs are difficult to design and implement and they are presented in Section 6.5.

2.3.12 Using Ontologies

Ontologies are unavoidable when dealing with agents. OMAS ontologies are formalized with the MOSS knowledge representation language, a powerful but complex frame language. MOSS is presented in a separate documents: MOSS user's Manual.

Chapter 3

The OMAS IDE

Contents

This chapter describes the OMAS IDE (Interactive Debugging Environment) and the features associated with the OMAS panel controlling it. The IDE is used to exercise agents and to visualize what is happening when the MAS is working. Its working is illustrated on the factorial example described in the second part of Tutorial 1 (Section 2.1.6).

3.1 OMAS Control Panel

The OMAS control panel has a number of features, associated with the various buttons. It allows the user to examine the state of the various local agents (statically) and trace how they interact (dynamically). Typically, we want to know

- How many agents we have
- What skills have the agents
- What is the current state of one agent
- What are the messages available
- What concept are recorded in the ontology of an agent

We want to do things to exercise the agents

- Create or modify a message
- Send a message

We want to monitor execution

- View the messages being exchanged
- View the messages an agent has received

- Trace the messages being exchanged
- Monitor the behavior of an agent

3.1.1 Examining Agents

Viewing the List of Agents

The local agents[1] of the application appear in the right pane of the control panel. Fig.3.1 displays the four agents of the factorial example. Note that the name of each agent is prefixed by SA_ for Service Agent.

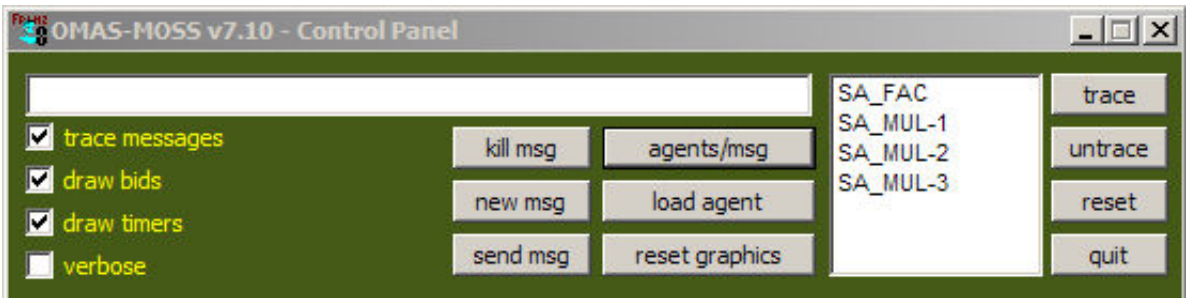

Figure 3.1: Agents from the factorial application

Viewing a Particular Agent

Clicking on one of the agents in the display opens the agent window that appears right below the control panel (Fig.3.2).

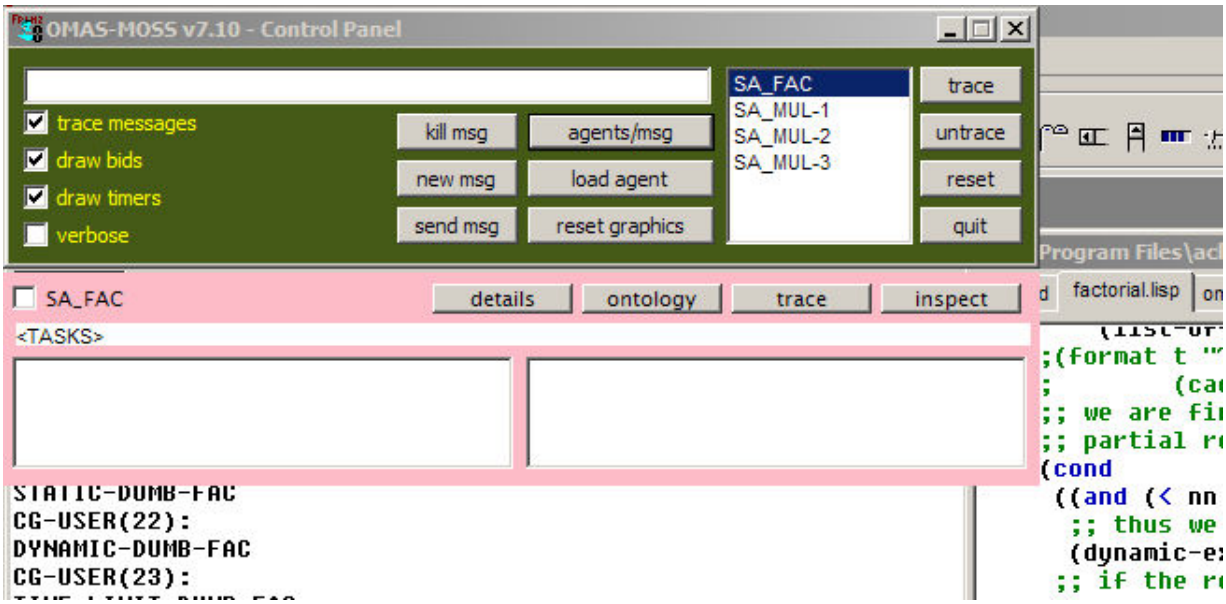

Figure 3.2: Agent window

[1] A local agent is one that executes on the local host.

The agent window contains two panes and several buttons. The panes will display incoming and outgoing messages. The buttons are:

- **details** that gives information about the internal state of the agent (Fig.3.3)
- **ontology** that displays the agent ontology (none in the case of the factorial example)
- **trace** for tracing the agent
- **export**, previously inspect, reserved to persistent agents for exporting the content of the ontology and knowledge base.

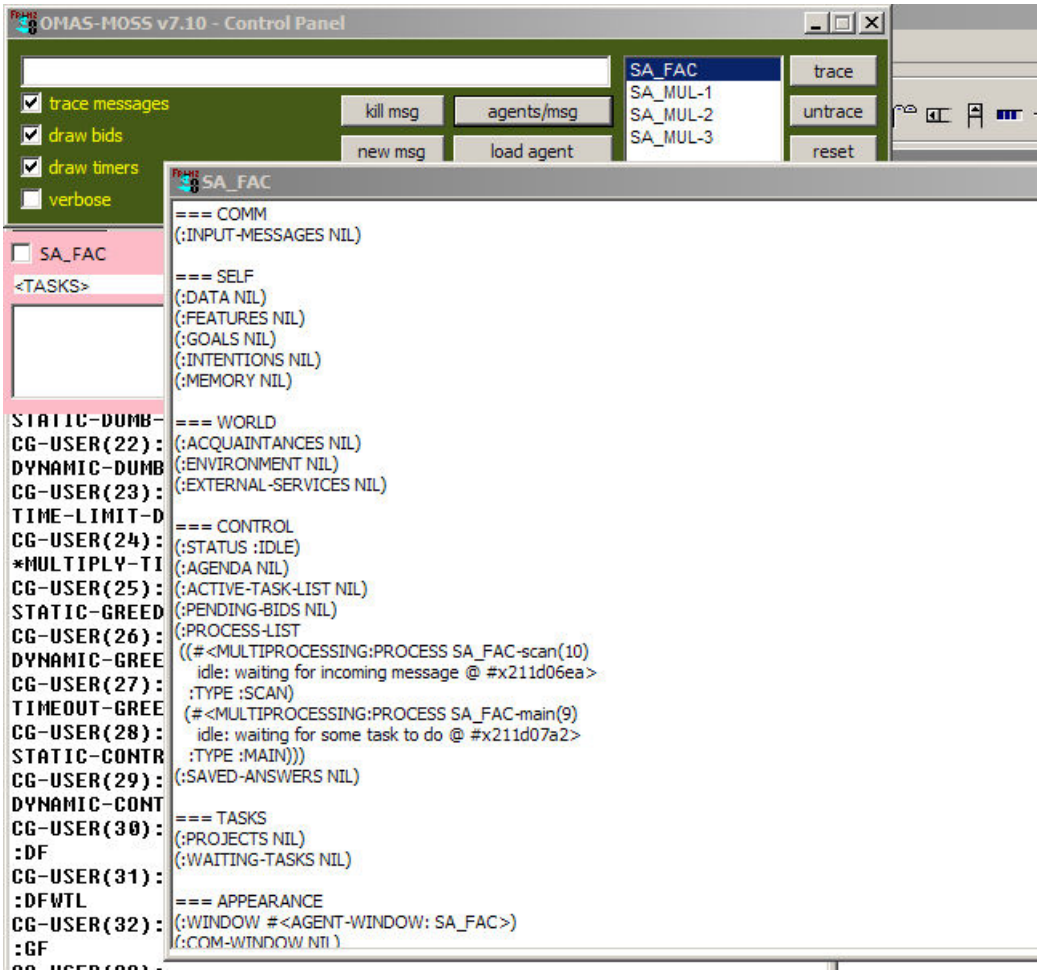

Figure 3.3: Some details of the content of the factorial agent

3.1.2 Creating and Sending Messages

Viewing the List of Available Messages

When some messages have already been defined, it is possible to view them by clicking the `agents/msg` button that switches between the list of agents and the list of messages (Fig.3.4).

Of course the available messages must have been predefined in the application file (usually by means of the `defmessage` macro in the Z-messages.lisp file).

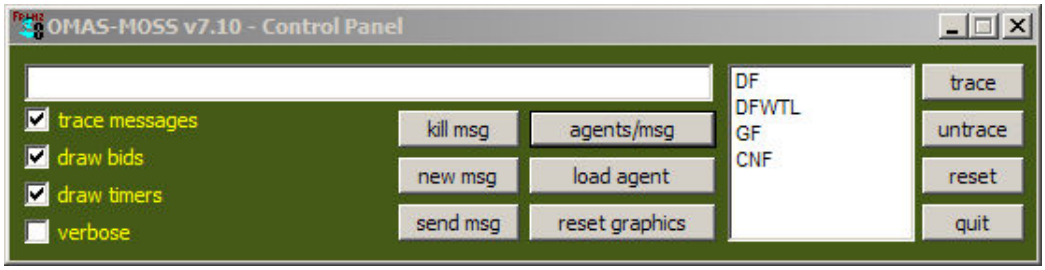

Figure 3.4: List of messages showing four predefined messages

Examining a Message

Selecting and examining a message is done by double clicking on one of the messages from the list, e.g. DF. A message editing window appears then as shown Fig.3.5.

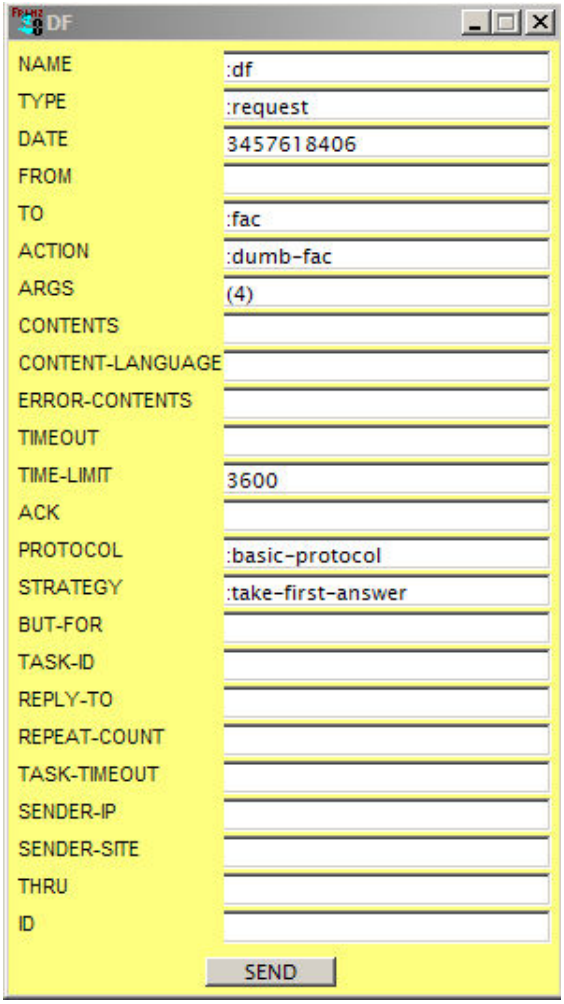

Figure 3.5: Message editing window showing the content of the DF (dumb factorial) message

The message shown Fig.3.5 is a request message, the sender is the user, the receiver is the agent :FAC, the requested action is :dumb-fac (a dumb way of computlng factorial), the list of arguments contains a single argument: 4. The other pieces of information have been added by OMAS and are

irrelevant here.

Creating a Message

A message can be created by clicking the `new msg` button. An empty message editing window appears, the fields of which can be filled by the user. The minimal information required to obtain a message similar to that shown Fig.3.5 is TYPE, TO, ACTION and ARGS, the rest is optional.

Sending a Message

There are two ways of sending a message:

- clicking on the `send` button of a message editing window (Fig.3.5)
- selecting a message in the message list and clicking on the `send meg` button of the control panel.

3.1.3 Monitoring Messages

It is important to be able to visualize exchanges of messages. This is done through the graphics window.

Clicking the reset graphics button of the control panel wakes up a graphics window that appears at the right side of the control panel. By default the graphics window displays life lines for the agents located on our machine (and also in more complex applications other agents of the platform known because they have sent messages). What is displayed in the graphics window is controlled by the three top left check boxes of the control panel. For example, if we select the DF message in the message list and send it by clicking the send msg button, we see the exchanges of messages (Fig.3.6).

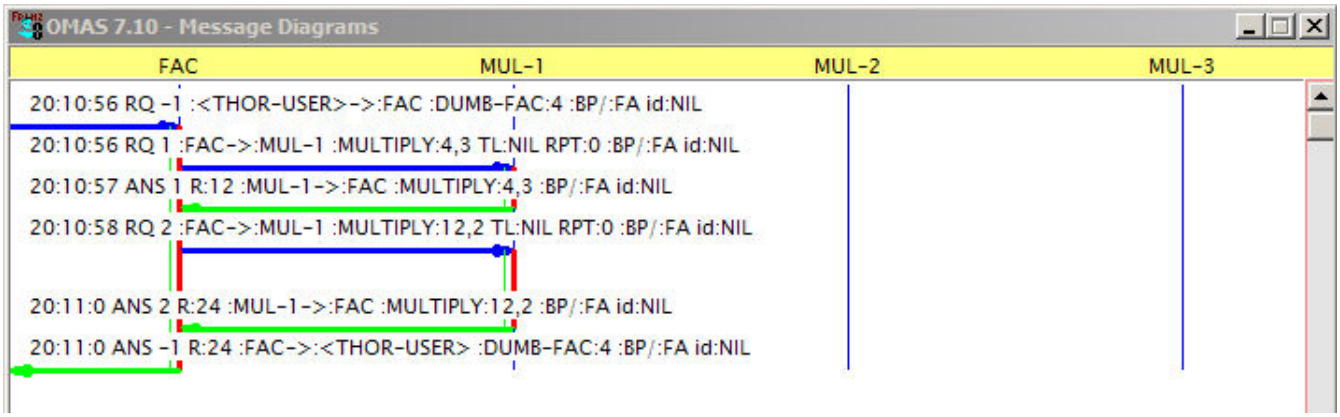

Figure 3.6: Monitoring the exchanges after sending the DF message

The user sends the DF message to the FAC agent. The FAC agent in turn sends a message asking the MUL-1 agent to multiply 4 and 3 (blue request message), waits until the answer comes back (green arrow), then sends a request asking to multiply 2 and 12 (previous result), waits for the answer and returns it to the user (who by convention resides at the left of the window).

We can notice several things:

- Each message has a color (blue for requests and green for answers).
- Each message has an associated text summary of the content of the message, starting with the time at which it was sent. Unchecking the `trace messages` check box in the control panel removes the text associated with the messages.

- The life line of an agent becomes red when the agent is busy (i.e. does at least one thing).
- At the left of an agent life line, there is a light green vertical line. This indicates a timer associated with the task of the agent (by default the maximal duration of a task is set to one hour [2]). Unchecking the `draw timers` check box in the control panel removes the timer lines.

3.1.4 Monitoring an Agent Execution

Monitoring an agent execution is the same as tracing an agent. To do so, one must select the agent in the list of agents and click on the trace button. Then, to be able to visualize the behavior of an agent, on must check the verbose check box, which opens a text window underneath the graphics window. One then can send a message and view what the traced agent is doing (Fig.3.7).

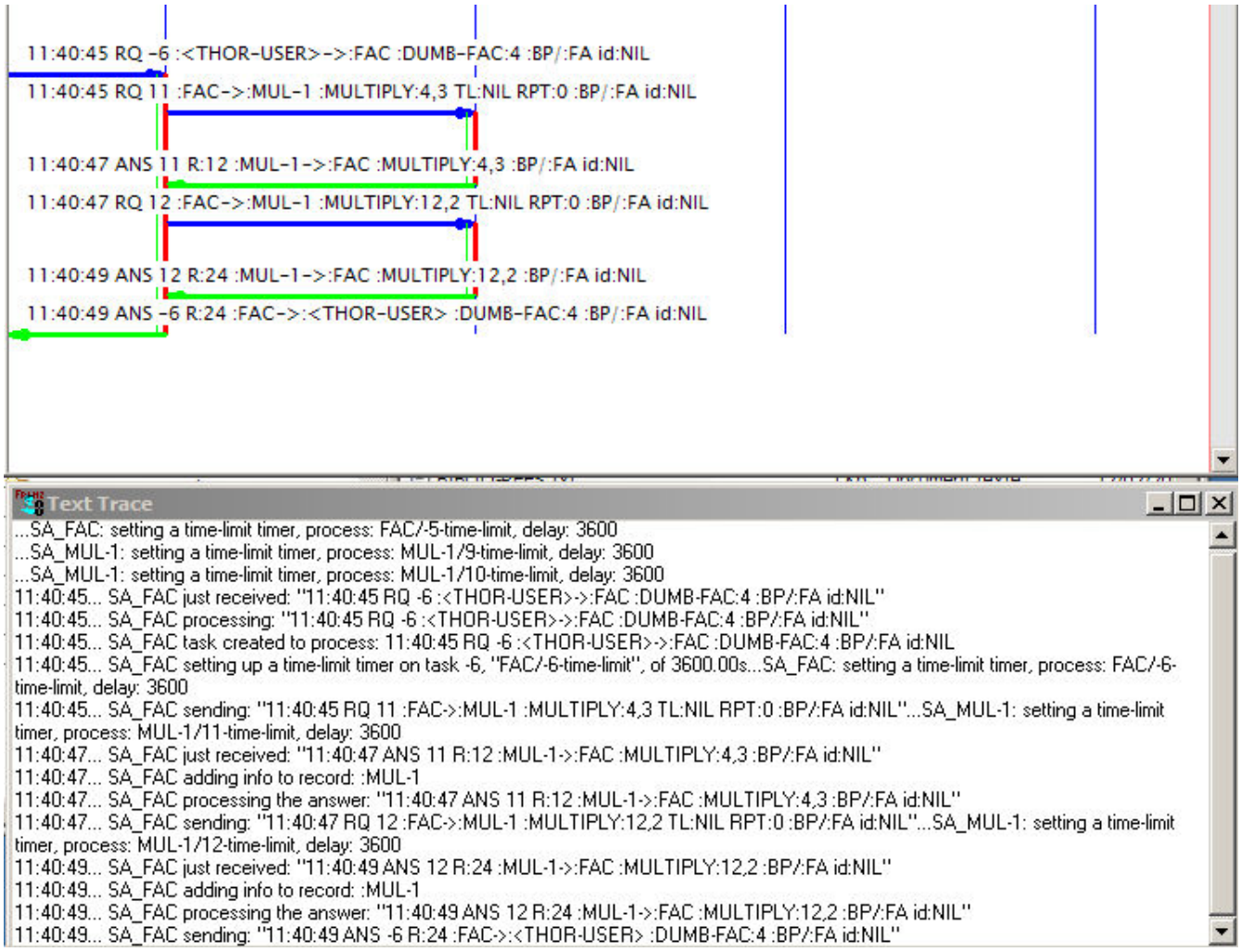

Figure 3.7: Monitoring the execution of the factorial agent

The Text Trace window show the following steps:

- FAC received a message from the user asking to compute 4! with the dumb approach
- it processed the user message

[2]This means that the task is aborted if not completed within one hour, which prevents the system to keep orphan threads when the corresponding agent is waiting for some event that does not happen

- it created a task
- ... and a timer with a delay of 1 hour (3600 seconds)
- then it sends a message to MUL-1
- etc.

Such a trace may be valuable, but is in general difficult to exploit.

3.1.5 Miscellaneous Control Buttons

Control Panel

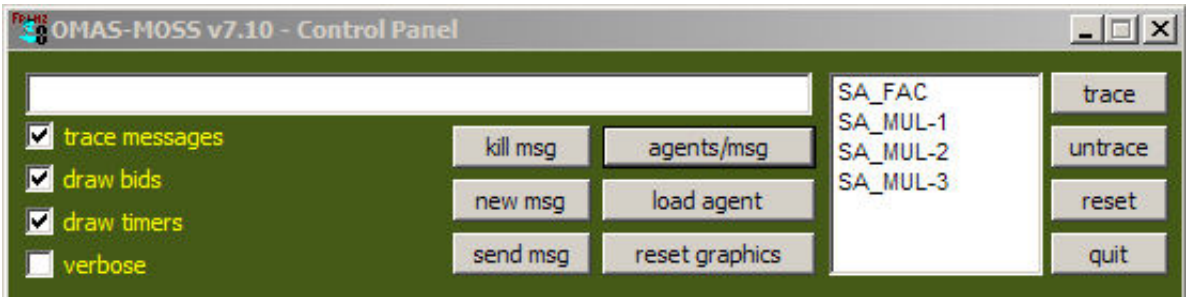

Figure 3.8: Agents from the factorial application

Let us summarize the role of the control buttons appearing on the control panel.

Buttons at the right hand-side

Trace	Allows printing a trace to follow what an agent is doing in a special `Text Trace window` (the verbose option check box should be checked). The agent is selected from the scrolling list
Untrace	Cancel the trace for an agent
Reset	Resets the system by reinitializing the various processes
Quit	Quits the application

Check boxes on the left hand-side

Trace messages	Toggles the text on and off on the Message Diagram
Draw bids	Draws (or not) the bids when the Contract Net protocol is used
Draw timers	Draws the life-line for the timers, when checked
Verbose	Prints a detailed trace of what is happening

Buttons on the center left hand-side

Kill msg	Removes the selected message from the list of messages
New msg	Creates a new message (a message box appears)
Send msg	Sends the selected message

Buttons on the center right hand-side

Agents/msg	Toggles between showing the agents or the predefined messages
Load agent	Allows to load a new agent (provided its files have been previously defined)
Reset graphics	Resets the graphics trace

Agent Window

Additional buttons have been added to the agent window in starting with OMAS v10 to cope with persistent agents.

Figure 3.9: Agent window

Let us summarize the role of the buttons of an agent window.

Top row

restore	Allow reloading a checkpointed file for persistent agents
details	Opens a window giving some details on the content of the agent
trace	toggles tracing/untracing an agent (coupled with the verbose check box of the control panel)
ontology	Opens a window showing the content of the associated ontology (all agents do not have necessarily an ontology)
checkpoint	For persistent agents dumps the content of the ontology and knowledge base into a text file to save the world in case the system later crashes and looses the content of the database
export	For persistent agents builds a textual ontology file exporting the content of the current ontology and knowledge base. Limited to the current context

Second row

tasks	displays the list of tasks being executed by a given agent (transient display)

Chapter 4

Architecture

Contents

This chapter describes the architecture of the OMAS platform. The first section deals with the global architecture of an application. The second section deals with the different models of agents offered by OMAS.

4.1 Global Architecture

A single machine may support several agents, in which case all agents reside in the same Lisp environment (whether MCL on MacOS or ACL on Windows). This fits in particular the situation in which a Personal Assistant (PA) has a staff of technical agents. Such a configuration is called a **local coterie**.

Local coteries are grouped into a larger coterie, supported by a network of several machines (Fig.4.1). A local coterie does not necessarily contain a personal assistant agent, but may rather shelter one or more service agents.

Within a coterie, all inter-agent messages are broadcast (using the UDP protocol). A coterie is bound by the physical connections of a network (number of adjacent loops a broadcast message can travel; usually a single loop). An OMAS *platform* may contain several coteries located on distant LAN loops.

Inter-coterie messages use TCP or HTTP. When connected to a foreign platform the exchange protocol (agent communication language) may be FIPA compliant. In other words, an OMAS platform can be a platform in the FIPA sense.

Within a coterie (platform), agents have unique names (assigned freely by the designer of the application).

Local Coteries

All agents are independent. They have their own name space, different from that of the coterie and different from the OMAS name space. The code defining an agent, its skills, goals, ontology, tasks or dialogs (for Personal Assistants), is contained in separate files.

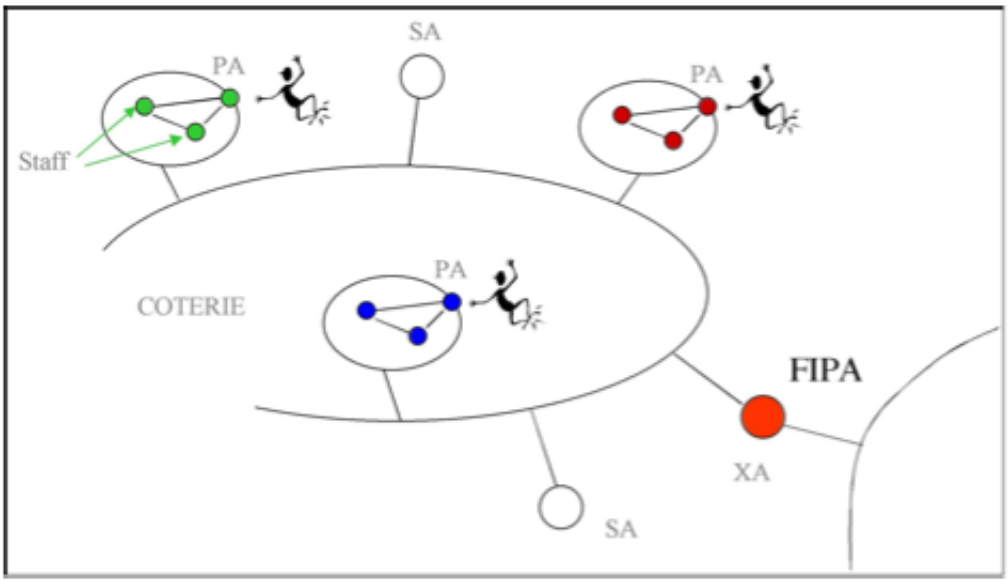

Figure 4.1: Coterie showing local coteries and inter-coterie link

A local coterie contains *invisible agents* namely, a *Spy agent* and a *Manager Agent*[1] .

4.2 Models of Agents

OMAS agents follow several models:

- Service Agents (SA)
- Personal Assistants (PA)
- Transfer Agents or Postmen (XA)
- Inferer Agents

They are briefly presented in the following paragraphs and described in the following chapters.

4.2.1 Service Agents

Service Agents are regular agents that provide a set of services. They can be thought of as Web Services with the difference that they may not answer, even if they can do the job. Their structure is shown Fig. 4.2.

The various parts of the structure of an agent are:

- net interface, handling message and log;
- skills, containing the skills corresponding to tasks the agent can do;
- world, containing a model of the world, namely a description of other agents obtained through the processing of messages;
- tasks, a model of the current tasks being active in the system (currently unused);

[1]The manager agent is not activated in the current OMAS version

- ontology, containing the ontology and knowledge base;
- self, containing a self description, a description of skills and the agent memory;
- control, containing all the necessary internal structures for running the agent.

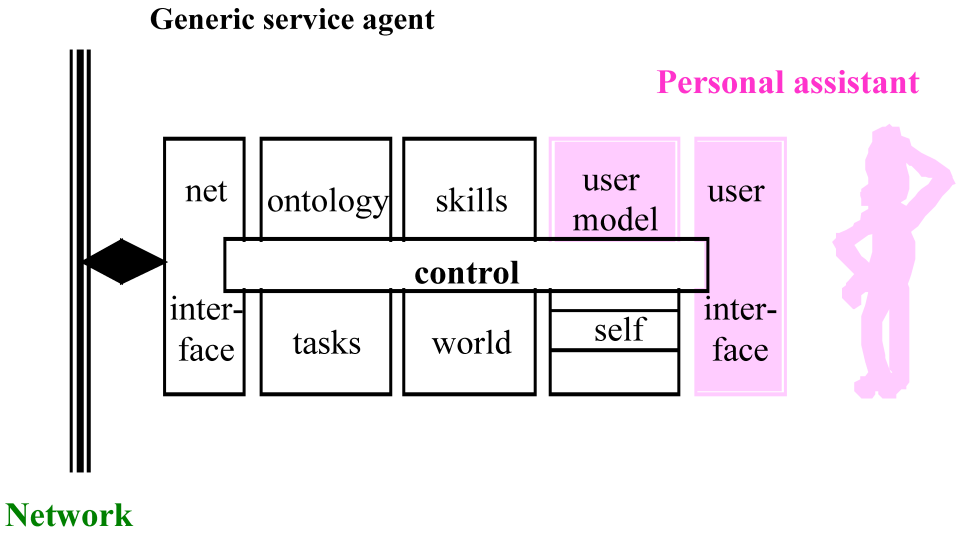

Figure 4.2: The structure of a Service Agent / Personal Assistant (with grey additions)

4.2.2 Personal Assistant and Staff Agents

A Personal Assistant (PA) is a *digital butler* in the Negroponte's sense. Its job is to communicate with its master. To avoid too much complexity, a personal assistant has staff agents, i.e. agents that perform a specific task either directly or by using other agents of the coterie. Staff agents are devoted to a specific personal assistant and answer its requests and requests from agents of the same staff, but do not answer requests from other agents of the coterie or from foreign agents.

4.2.3 Transfer Agents (Postmen)

A Transfer Agent (XA) also called a *postman* is a gateway between an OMAS coterie and another remote coterie belonging to the same platform, or between a platform and a foreign platform, or between a platform and Web Services, etc. It is used to implement other protocols and encapsulate the corresponding services.

4.2.4 Inferer Agents

An Inferer Agent (IA) is an agent defined in terms of production rules. It is capable of inference using the rules. Currently the inference engine is limited to simple forward chaining. An important point (for some people) is that it does not contain any Lisp code.

Chapter 5

Service Agent

Contents

This chapter describes the structure of a Service Agent (SA), and various mechanisms like skills, memory, or goals. A service agent is a basic agent structure on top of which other agents are built. The chapter takes the point of view of a developer who wants to create an SA.

5.1 Structure

Creating an SA is done simply by using the defagent macro:

```
(defagent :OSCAR)
```

The macro can take several options:

Option	Usage
:context	specifies the version context for the MOSS objects (default is 0)
:hide	specifies if true that the agent remains hidden from the user. E.g. the agent handling the graphics window is hidden from the user. This is normally reserved for system agents that are not displayed to the user.
:language	specifies the agent language (default language is :EN)
:learning	from OMAS v10 is deprecated, not being very useful
:master	means that the agent is a Staff Agent serving a master (a Personal Assistant). The agent can only give answers to its master, i.e. only the master can send request to this agent. However, it can send messages to any other agent on the platform and receive answers directly. Note. An agent may have several masters. This may be convenient when a user has several PAs, e.g. speaking different languages.
:ontology-file	specifies the pathname of a file containing the ontology to be loaded
:package-nickname	specifies a nickname for the agent package, e.g. :AL
:persistency	specifies if true that the agent has a persistent store in which it keeps its ontology and knowledge base. Saving the initial ontology and knowledge base and reopening it on later connections is done automatically. In the ACL environment OMAS uses Allegrostore. In the MCL environment OMAS uses Woods database.
:version-graph	specifies the structure of the different versions of the ontology and the knowledge base (default is version 0)

5.2 Skills

An agent has skills corresponding to what it can do (not to be confused with goals corresponding to what it is planning to do).

5.2.1 The defskill macro

Creating a skill is done simply by using the defskill macro. In its simplest form:

```
(defskill :bye :OSCAR
   :static-fcn static-bye)
```

When receiving a message where the action is :bye the agent will execute the static-bye function defined as:

```
(defun static-bye (agent message {arg*}) <body>)
```

where agent points to OSCAR and message is the message just received. {arg*} represent optional arguments of the skill. Note that the full Lisp syntax can be used, namely: optional arguments, key arguments, rest argument, etc.

A skill has a number of possible options:

Option	Usage
acknowledge-fcn	user defined handler to process returned acknowledgments
bid-cost-fcn	user defined function for computing the cost of a bid
bid-quality-fcn	user defined function for computing the quality of a bid
bid-start-time-fcn	user define function for computing earliest start time
description	description of the skill
dynamic-fcn	handler for processing subtask answers
how-long-fcn	user defined function returning time needed to compute
how-long-left-fcn	user defined function returning time left to completion
name	name of the skill
preconditions	user defined function to check arguments
select-best-answer-fcn	user defined function to select contract-net answers
select-bid-fcn	user defined function to select contract-net bids (currently unused)
static-fcn	function executed when activating the skill for the first time
static-pattern	for checking static arg types
time-limit-fcn	handler for processing time limit interrupts
timeout-handler	handler for processing timeouts

5.2.2 Acknowledge Option

Sometimes it is useful to know if a message has reached some agent, in particlar when the answer might take a long time. For such cases, we can use the :ack option in a message.

Definition

When an agent receives a message containing the :ack option, it then returns an acknowledge message before trying to process the received message. Sending an acknowledge message does not imply that the agent will process or answer the received message. It simply indicates that the message was received.

Example

```
(send-subtask :to :joe :action :confirm-appointment :args ("Tuesday 10 am") :ack t)
```

When the acknowledge message returns we may want to process it independently from the task dynamic part. To do so we may define an acknowledge handler.

Specifying an Acknowledge Handler

This is done using the `:acknowledge-fcn` option of the `defskill` macro.

```
(defskill :CONFIRM-APPOINTMENT TOM.
  :static-fcn static-confirm-appointment
  :dynamic-fcn dynamic-confirm-appointment
  :acknowledge-fcn acknowledge-confirm-appointment
  )

(defun acknowledge-confirm-appointment (agent message info)
  "called when receiving an acknowledge message"
  (declare (ignore message))
  ;; memorize whatever info was returned
  (remember agent info :appointment-request-acknowledgement))
```

The arguments to an acknowledge handler are the agent, the task message and the arguments of the subtask message.

5.2.3 Bid Cost Option

This option allows the user to specify a function for computing the cost of doing a task. For example if the agent is a bookstore and it receives a request for a book, then the cost could be the price of the book, plus a shipment cost, plus a service charge.

Example:

```
(defskill :BOOK-REQUEST :OSCAR
  :static-fcn static-book-request
  :bid-cost-fcn bid-cost-book-request
  )

(defun bid-cost-book-request (agent args)
  "called when when computing the cost associated to a bid"
  (declare (ignore message))
  (let ((book (access `("book" ("title" :is ,(car args))))))
  ;; if we have the book get its price
  (if book (car (send (car book) '=get "price"))
  ;; otherwise put large number
    1000000))
```

5.2.4 Bid Quality Option

This option allows the user to specify a function for computing the quality of the result of a task on a scale 0 to 100. Default is 100. This can be used when the task is a computation using an approximate method for example.

5.2.5 Bid Start Time Option

This option allows the user to specify a function for computing the number of seconds before a task can start if granted. Default is 0, since whenever an agent receives a task it starts a new thread to execute it.

5.2.6 Dynamic Option

The option is a handler that receives the answers of subcontracted tasks. If agents send answers then the handler is activated. If nobody sends an answer, then we must provide a timeout handler.

An example was given with the factorial agent in the tutorials:

```
(defun dynamic-dumb-fac (agent message answer)
  "this function is called whenever we get a result from a subtask. This ~
      approach is not particularly clever, since the computation is linear ~
      and uses the same multiplying agent, i.e., MUL-1."
  (declare (ignore message))
  (let ((nn (env-get agent :n)))
    ;; if the recorded value is 1 or less, then we are through
    (if (< nn 2)
      ;; thus we do a final exit (in fact we should never go there ??
      (dynamic-exit agent answer)
      ;; otherwise we multiply the answer with the next high number
      ;; creating a subtask
      (progn
        (send-subtask agent :to :MUL-1 :action :multiply
                      :args (list answer nn))
        ;; update environment
        (env-set agent (1- nn) :n)
        ;; then the function returns a value to nobody in particular
        answer
        ))))
```

The answer from a subcontracted agent *is passed directly to the handler*. Event processing and message dispatching are done by the platform. An agent can do several factorial computations in parallel, there is no danger of mixing the data.

5.2.7 How-long Option

This option allows the user to specify a function for computing the length in seconds a task will take to execute. This could be difficult for a standard task. However, it can be used in the case of a PA when the master has to answer a question. The PA could evaluate the time the master will take to answer the question.

5.2.8 How Long Left Option

This option allows the user to specify a function for computing the length in seconds a task will take to complete. Again, this is in general a difficult question.

5.2.9 Preconditions Option

Sometimes it is necessary to dynamically check the arguments of a message before deciding to consider it or not. This can be done with the `:preconditions` option of the `defskill` macro.

Definition

Preconditions are a test on the arguments associated with a skill.

Mechanism

When a precondition function has been defined, it is executed prior to taking into account the corresponding subtask. If it returns nil, then the subtask is disregarded.

Specifying a Preconditions Function

```
(defskill :MULTIPLY MUL-1
  :static-fcn multiply
  :preconditions-fcn preconditions-multiply
  )

(defun preconditions-multiply (agent environment n1 n2)
  "called before triggering the skill?"
  (declare (ignore agent environment))
  (and (integerp n1) (integerp n2)))
```

Preconditions are not restricted to checking the type of the arguments. One could imagine that the multiplying agent does not know how to multiply large numbers for example.

5.2.10 Select Best Answer Option

Definition

When a subtask was broadcast and the strategy is to collect answers, then we expect to receive several answers. The answers are collected as a list of messages and when there are enough answers (collect first n answers) or the timeout occurs, OMAS checks for the existence of a user-defined select-best-answer function, and if there applies it to the list of answers.

Mechanism

The process is the following:

1. If there is a user-defined select-best-answer function, then it is applied to the list of messages.
   (a) if the result is NIL, then the task is aborted
   (b) if the function returns a list containing a message and the associated value (whatever it is), then the dynamic skill is called with the message argument set to the best answer message and the third argument (after agent and message) set to the corresponding best value.
2. if there is no user-defined function, then dynamic part of the skill is called with the :unprocessed tag as the message argument, and the entire list of answer messages (including errors) as the third argument.

Specifying a Preconditions Function

```
(defskill :CNET FAC
  :static-fcn static-fac
  :select-best-answer-fcn best-answer-fac
  )

(defun select-best-answer-fac (agent answer-list)
  "select best answer"
```

```
(declare (ignore agent))
(let (good-answers best-list)
  ;; remove error messages
  (setq good-answers
    (remove nil
     (mapcar #'(lambda (xx) (if (eql (omas::contents xx) :error) nil xx))
            answer-list)))
   ;; sort the messages according to values (function to provide)
   (setq best-list (sort ll #'compare-values
                  :key #'(lambda (xx)(omas::contents xx))))
;; return best message and value
(list (car best-list) (omas:: contents (car best-list)))))
```

5.2.11 Select Bid Option

Currently unused, i.e. we cannot specify a user function.

Bids are compared as follows: if we have two messages msg1 and msg2, each message content must be a list (start-time delay quality cost rigidity). OMAS compares first the cost, then the completion time (start-time + delay), then the quality. If msg2 is better than msg1 returns nil.

5.2.12 Static Function Option

The static option is required and corresponds to the function applied to the incoming message. Arguments are agent, message and zero or more arguments with the syntax of lisp functions.

5.2.13 Time Limit Option

If the timeout applies to the sender of the message, the time limit applies to the receiver of the message.

Definition

A time limit is the maximum time given to an agent for executing a task.

Specifying a Time Limit

A time-limit can be specified in a message by using the `:time-limit` option, e.g. specifying a time-limit of 20 seconds for a subtask.

```
(send-subtask agent :to (answering-agent agent) :action :multiply
                    :args (list answer (cadr (member :res environment)))
                    :time-limit 20)
```

Default Behavior

By default when a time limit is reached OMAS gives two more possibilities (extending the time-limit twice). On the third retry, the task is aborted.

Specifying a Time Limit Handler

This is done by using the `:time-limit-fcn` option of the `defskill` macro.

```
(defskill :fast-fac fac
  :static-fcn static-fast-fac
  :dynamic-fcn dynamic-fast-fac
  :time-limit-fcn time-limit-fast-fac)

(defun time-limit-fast-fac (agent environment message)
  "function for handling time-limit errors"
  (declare (ignore environment)
  (format t "~&Warning; we went over time to process subtask ~S"
            (omas::task-id message))
  (abort-current-task agent))
```

Normally, the user aborts the task in the handler. However, when this is not the case, the default behavior takes over, i.e., two retries are allowed.

Usage

Of course a time limit option can be used to tell an agent that its time to execute a specific skill is limited. Thus, if the agent knows that it cannot execute the subtask within the specified time, it can avoid undertaking it. In practice this is difficult to evaluate. Hence the time limit mechanism is rather used by the system internally to avoid orphaned tasks. OMAS associates a time limit of 1 hour to each task. Thus, after 1 hour, the task is aborted, cleaning the agent environment.

Another possible use is one involving human intervention. When a subtask must be solved by a human, then the 1 hour deadline may be too short and it is possible to specify a longer time-limit for executing the subtask.

5.2.14 Timeout Option

Definition

A timeout specifies the maximum time an agent is willing to wait for an answer before taking action.

Note: A timeout thus is used by the sending agent and has no meaning for the receiving agent.

Types of Timeouts

We can distinguish two kinds of timeouts:

- A normal timeout that constitutes a warning and gives the choice to wait some more time after having taken some possible corrective action,
- A severe timeout when all solutions have been exhausted and no answer was returned.

Specifying a Timeout

When sending a subtask A timeout limit can be specified in the send-subtask function by means of the `:timeout` parameter. E.g., within the dynamic part of the skill for computing a factorial, we can insert a timeout in the subtask function call:

```
(send-subtask agent :to (answering-agent agent) :action :multiply
                    :args (list answer (cadr (member :res environment)))
                    :timeout 2.5)
```

OMAS will create a timer and the subtask will be aborted (default behavior if no timeout handler has been defined) after 2.5 seconds. By default the main task itself (i.e. the one that sent the message) will be aborted.

Providing a Timeout Handler Specifying a user-defined timeout handler can be done by using the `:timeout-handler` option of the `defskill` macro. E.g.

```
(defskill :fast-fac-with-timeout fac
  :static-fcn static-fast-fac-with-timeout
  :dynamic-fcn dynamic-fast-fac-with-timeout
  :timeout-handler timeout-fast-fac-with-timeout)

(defun timeout-fast-fac-with-timeout (agent message)
  "function for handling timeout errors"
  (case (omas::action message)
    (multiply
     ;; if a multiply subtask is timed out, then reallocate to another agent
     ;; by simply doing another broadcast
     (send-subtask agent :to :ALL :action :multiply
                   :args (omas::args message)
                   :protocol :contract-net
                   :timeout *multiply-timeout*)
     ;; exit
     :done
     )
    (otherwise
     ;; for any other skill we let the system process the timeout condition
     :unprocessed))
  )
```

On timeout the handler is called and runs.

The handler arguments are the agent structure and the message that was sent. Parts of the message can be accessed by using (yet undocumented) internal OMAS functions. In the example `omas::action` retrieved the action specification of the message, and if the action is `:multiply` executes a new broadcast.

Note An important point is that there exists but a single timeout-handler within a given task, which means that it could be used for different types of subtasks.

Default Timeouts in OMAS

Standard messages: No timeout on standard messages.

Broadcast messages: The default broadcast timeout is 3 seconds defined by the internal global parameter contained in the global parameter object omas::*omas*:

```
(omas::broadcast-default-timeout omas::*omas*)
```

Contract-Net call-for-bids timeout: The default Contract-Net call-for-bids timeout is 0.5 second defined by the internal parameter:

```
(omas::default-call-for-bids-timeout-delay omas::*omas*)
```

5.2.15 Predefined Skills

An agent when created has predefined skills:

- HELLO: when sending an HELLO request message to an agent without arguments, the agent answers "Hello from <name of the agent>.
- PING: when sending a PING request message to an agent, the agent answers with its name key.
- SEND: used when one wants to ask an agent to send a message to other agents. It has a static and dynamic part and a timeout handler.
- FIND: used to ask an agent to find an individual corresponding to a particular concept and described by a list of words in natural language.
- SEND-MESSAGE: allows any agent to send a request message instantaneously. Called following a message like
  :from agent :to agent :type :internal :action :send-message :args ' (:type :inform :to :mul-1 :action :mutilply :args '(4 5) :delay 3)

5.3 Goals

5.3.1 Overview

Goals are defined so that an agent can display a particular behavior.

A goal is constructed as an object structure and kept inside the agent memory. A goal has different parameters like a type (cyclic, one-shot), a period, an activation date, an expiration date, an importance, an urgency, a status, or a script.

Scripts

Goals use scripts. A script represents a plan to achieve the goal. A script is a function that is called on the particular agent. It produces a list of internal messages that are inserted into the agent mailbox when the goal fires. For example a skill may produce an internal request.

Energy (not available yet)

Associated with a goal is a level of energy on a 0-100 scale together with a threshold. As long as the level is below the threshold, the goal is not activated. A level of 0 means that the goal is never activated a level of more than 100 means that the goal is always activated. The exact mechanism for changing the energy level is yet unspecified. The idea is related to the activation energy level of Patty Maes, or to the stressed agents (eco-resolution of reactive agents).

5.3.2 Detailed Goal Mechanism

A goal is represented within an agent by a structure in its memory (part of the *self* model). The goal structure contains the parameters that control the firing of the goal as well as optional functions that can modify the normal control mechanism (for increased flexibility).

Goal Structure

A goal object has a number of properties:

```
GOAL
   name                 (name)
   type                 (cyclic, one-shot,...)
```

```
mode                  (rigid, flexible/allows activation level mechanism)
period                (period for a cyclic goal)
expiration-date       (date at which it dies)
expiration-delay      (delay after which it dies)
expiration-process    (timer process that will kill the goal)
importance            (high, medium, low)
urgency               (high, medium, low)
status                (waiting, active, dead,...) may not be useful with activation
activation-date       (date at which the goal should fire)
activation-level      (on a 0-100 scale)
activation-threshold  (value above which the goal is activate)
activation-change-fcn (fcn that change the activation level at each cycle)
goal-enable-fcn       (function, can be included in the static skill)
script                (function that produces a list of messages)
```

The various properties of the goal object have the following meaning and usage.

- *name* records the name of the goal (e.g., :WAKE-UP)
- *type* describes the type of the goal that can be
  - *cyclic*, in which case it will be checked periodically according to the period delay specified in the period argument
  - *one-shot*, in which case it will be fired only once after which its status will be set to dead.

  By default goals are of the one-shot type.
- *mode* can be rigid or flexible. A rigid goal is controlled by the clock. A flexible goal is controlled by its level of energy (not yet implemented).
- *importance*, *urgency* are currently unused.
- *status* gives the status of the goal as a basic mechanism. The status may be `:waiting`, `:active`, or `:dead`. It may not be necessary when one uses a more sophisticated mechanism like the level of energy (de facto when the level of energy stays below the threshold, then the goal is waiting).
- *activation-date* date at which the goal is activated (expressed in universal-time).
- *activation-level* is the current level of activation energy
- *activation-threshold* is the level at which the goal is fired.
- *activation-change-fcn* is a function that is called at each cycle for updating the level of activation. It is called by the system but defined by the user.
- *goal-enable-fcn* is a user defined function, called the goal is ready to fire and can prevent it from firing.
- *script* is the name of a function expressing the scenario implementing the goal, i.e., returning a detailed plan expressed as a series of internal messages.

The script is currently expressed as a function of one argument (the current agent) that should return a list of messages to be sent when the goal is fired. However, this may prove insufficient, and a script language should be defined.

Control Mechanism

When a goal is created two timers are also created. The first timer, an activation timer, will fire when it is time to activate the goal, the second timer is a time limit. It is created if the goal has an expiration date. When it fires the goal will be disabled.

When the goal must be repeated periodically, the launching timer will be rearmed to fire after the length of the specified period.

If a goal is disabled, then its status is set to `:dead`. However the goal structure is not removed from the agent memory.

Advanced Mechanisms (not implemented yet)

Advanced mechanisms are related to the mechanism of *activation energy*. The main idea is to give each goal a level of energy and to modify it periodically by means of a special timer that will activate the goal-enable-fcn. When the energy level reaches the activation threshold, then the corresponding action is inserted into input-messages queue of the agent (mailbox). Two problems remain to be solved: (i) how does the energy level changes at each cycle; and (ii) what do we do to the energy level when the goal fires.

5.3.3 Implementation

Creating Goals

A goal is created by means of the make-goal function or the defgoal macro. Once it is created it becomes active.

make-goal *goal-name agent-name &key (mode :rigid) (type :1-shot) (period 10) expiration-date expiration-delay importance urgency activation-date (activation-delay 0) (activation-level 50) (activation-threshold 50) (status :waiting) (goal-enable-fcn 'agent-goal-always-true) script* **function**

Required Arguments

(`goal-name` *name (keyword) chosen to specify the goal*)
(`agent-name` *name (keyword) of the agent concerned by the goal*)

Optional Keyword Arguments

(`mode` *activation mode, i.e., :rigid or :flexible (default is :rigid)*)
(`type` *type of goal :cyclic :1-shot (default is :1-shot)*)
(`period` *period for cyclic goals (default is 20)*)
(`expiration-date` *date at which the goal dies, i.e., becomes inactive*)
(`expiration-delay` *delay after which the goal dies, i.e., becomes inactive*)
(`importance` *on a 1-100 scale (currently unused)*)
(`urgency` *on a 1-100 scale (currently unused)*)
(`activation-date` *date at which the goal should fire (default is now)*)
(`activation-delay` *delay after which the goal should fire (default is now)*)
(`activation-level` *on a 1-100 scale (default is 50)*)
(`activation-threshold` *on a 1-100 scale (default is 50)*)
(`status` *:waiting, :active, :dead,... may not be useful with activation*)
(`goal-enable-function` *goal allowing goal to fire (if non nil return). Default is* agent-goal-always-true *that returns t*)

(`script` *function that takes the agent as an argument and must return a list of messages*)

defgoal *goal-name agent-name &key (mode :rigid) (type :1-shot) (period 10) expiration-date expiration-delay importance urgency activation-date (activation-delay 0) (activation-level 50) (activation-threshold 50) (status :waiting) (goal-enable-fcn 'agent-goal-always-true) script* **macro**

Calls the make-goal function.

Example

The following definition creates a one-shot goal named :send-action for agent :BOSS, starting immediately, and executing the send-action script.

```
(defgoal :send-action :boss :script send-action)
```

expands to:

```
(MAKE-GOAL :SEND-ACTION :BOSS
           :MODE :RIGID
           :TYPE :1-SHOT
           :GOAL-ENABLE-FCN 'AGENT-GOAL-ALWAYS-TRUE
           :SCRIPT 'SEND-ACTION)
```

The make-goal function creates the goal structure, and installs it with the agent. It returns a list of the goal name and the agent name.

5.3.4 Activating Goals

Goals are activated by their activation timer. When it fires the timer calls the agent-run-goal function.

Processing a Rigid Goal

The agent-run-goal function checks whether the goal is dead or not. If dead it simply does nothing. If not dead, it checks whether the goal has a goal-enable function; if yes, then runs it. If the result is non NIL, it then executes the goal (currently putting the goal messages into the input-messages queue of the agent). Otherwise, does nothing or simply rearms the activation timer in case of a periodical goal.

Warning The list of messages corresponding to the goal are inserted into the mailbox of the agent. **Thus, the messages may not be processed in the order in which they were written**.

Processing a Flexible Goal

Unclear as of now.

5.3.5 Examples

The following paragraph contains the description of a simple test that can be easily performed to check the functioning of the goal mechanism. One defines a simple agent with a simple skill :PRINT that prints the current time (value of the clock).

If you want to exercise the following examples, you should be extremely careful in specifying the names of the skills, of the goals and the associated functions.

Agent, Skill and Scenario

We define a single TEST agent that will receive an :inform message asking to print something (PRINT skill).

```
? (defagent :TEST :redefine t)
#<AGENT TEST>

? (defskill :PRINT :TEST
  :static-fcn static-TEST-PRINT)
(:PRINT TEST)

? (defun static-test-print (agent environment)
  "simple skill that prints a message whenever the agent is called"
  (declare (ignore environment))
  (format t "~&Hello, my name is TEST, and current time is ~S"
            (time-string (get-universal-time)))
  (static-exit agent nil))
```

STATIC-TEST-PRINT

Simple Rigid Goal

Let us set a 1-shot goal to be executed at t=3s.

First we must define the script: a function producing a list of messages that will be inserted into the agent agenda.

```
? (defun goal-print-1 (agent)
    (list (make-instance
           'omas::message :type :request
           :from (omas::key agent) :to (omas::key agent)
           :action :print
           :args nil
           :task-id :T0
           )))
GOAL-PRINT-1
```

Then we define (and activate) the goal itself:

```
? (make-goal 'PRINT-1 :TEST
     :activation-date  (+ (get-universal-time) 3) ; call at t=4 seconds
     :script           'goal-print-1)
```

This produces the following trace (verbose option, i.e. `omas::*omas-verbose*` set to true).

```
;=== make-goal; exp-delay: goal expires in NIL seconds
;= make-goal; time now: "18:16:45"; activation-date: "18:16:48"; activation-delay: 0;
 activation-time: "18:16:48"; setting goal process.
;= make-goal; time now: "18:16:45"; expiration process set.
#<GOAL PRINT-1: 1-SHOT AD:3425559408 ED:NIL AL:50 AT:50 S:WAITING>
;=== agent-run-goal; time now: "18:16:45"; delay: 3 seconds
?
```

After 3 seconds the following messages are is printed:

```
;= agent-run-goal: time now: "18:16:48"; goal-enable-fcn: OMAS::AGENT-GOAL-ALWAYS-TRUE

;=== agent-execute-goal; time now: "18:16:48";
;= message-list: (#<MESSAGE 18:16:48 :TEST :TEST :REQUEST :PRINT NIL NIL Tid::T0 :BASIC-PROTOC
...TEST: setting a time-limit timer, process: TEST/:T0-time-limit, delay: 3599

Hello, my name is TEST, and current time is "18:16:49"
```

One can check that the status of the goal has been set to `:dead`
Running the goal again with `omas::*omas-verbose*` set to nil produces the following trace:

```
? (make-goal 'PRINT-1 'TEST
     :activation-date  (+ (get-universal-time) 3) ; call after 3 seconds
     :script  'goal-print-1)
#<GOAL PRINT-1: 1-SHOT AD:3425559789 ED:NIL AL:50 AT:50 S:WAITING>
?
```

And, after 3 seconds:

```
Hello, my name is TEST, and current time is "18:23:9"
```

Periodical Rigid Goal

We use the same agent but a periodical goal.

```
? (defun goal-print-2 (agent)
    (list (make-instance 'omas::message :type :request
                         :from (omas::key agent) :to (omas::key agent)
                         :action :print
                         :args nil
                         :task-id :T1
                         )))
GOAL-PRINT-2

? (make-goal 'PRINT-2 'TEST
  :type             :cyclic
  :period           4           ; seconds
  :activation-date  (+ (get-universal-time) 3)   ; call at t=1 then every 3 seconds
  :script           'goal-print-2)
#<GOAL PRINT-2: CYCLIC AD:3425560329 ED:NIL AL:50 AT:50 S:WAITING>
```

creates the following goal (internal structure):

```
OMAS::NAME: PRINT-2
OMAS::MODE: :RIGID
TYPE: :CYCLIC
OMAS::PERIOD: 4
OMAS::EXPIRATION-DATE: NIL
OMAS::EXPIRATION-DELAY: NIL
OMAS::EXPIRATION-PROCESS: NIL
```

```
OMAS::IMPORTANCE: NIL
OMAS::URGENCY: NIL
OMAS::ACTIVATION-DATE: 3337144166
OMAS::ACTIVATION-DELAY: 0
OMAS::ACTIVATION-LEVEL: 50
OMAS::ACTIVATION-THRESHOLD: 50
OMAS::ACTIVATION-CHANGE-FCN: NIL
OMAS::STATUS: :ACTIVE
OMAS::GOAL-ENABLE-FCN: NIL
OMAS::SCRIPT: GOAL-PRINT-2
```

This produces the following trace:

```
? Hello, my name is TEST, and current time is "18:32:9"
?
Hello, my name is TEST, and current time is "18:32:13"
?
Hello, my name is TEST, and current time is "18:32:17"
?
Hello, my name is TEST, and current time is "18:32:21"
?
Hello, my name is TEST, and current time is "18:32:25"
?
Hello, my name is TEST, and current time is "18:32:29"
?
Hello, my name is TEST, and current time is "18:32:33"
?
Hello, my name is TEST, and current time is "18:32:37"
?
Hello, my name is TEST, and current time is "18:32:41"
```

To kill this goal, one first has to get the list of the goals of the agent TEST:

```
? (omas::goals test)
(#<GOAL PRINT-1: 1-SHOT AD:3425559408 ED:NIL AL:50 AT:50 S:DEAD>
 #<GOAL PRINT-1: 1-SHOT AD:3425559789 ED:NIL AL:50 AT:50 S:DEAD>
 #<GOAL PRINT-2: CYCLIC AD:3425560329 ED:NIL AL:50 AT:50 S:ACTIVE>)
```

... and kill the last one:

```
? (car (last *))
#<GOAL PRINT-2: CYCLIC AD:3425560329 ED:NIL AL:50 AT:50 S:ACTIVE>
? (omas::agent-kill-goal test *)
:GOAL-KILLED
```

Verify:

```
? (omas::goals test)
(#<GOAL PRINT-1: 1-SHOT AD:3425559408 ED:NIL AL:50 AT:50 S:DEAD>
 #<GOAL PRINT-1: 1-SHOT AD:3425559789 ED:NIL AL:50 AT:50 S:DEAD>)
```

The goal has been removed from the list of goals. A less drastic action would have been to set its status to `:dead`.

Combining Several Goals

In order to understand the trace more easily, we create several skills.

```
(defskill :PRINT :TEST
  :static-fcn static-TEST-PRINT)

(defskill :SHOUT :TEST
  :static-fcn static-TEST-SHOUT)

(defskill :YELL :TEST
  :static-fcn static-TEST-YELL)

(defun static-test-print (agent environment)
  "simple skill that prints a message whenever the agent is called"
  (declare (ignore environment))
  (format t "~&Hello, my name is ~A and time is ~A"
          (omas::name agent) (omas::time-string (get-universal-time)))
  (static-exit agent "*done*"))

(defun static-test-shout (agent environment)
  "simple skill that prints a message whenever the agent is called"
  (declare (ignore environment))
  (format t "~&SHOUTING, my name: ~A at time ~A"
          (omas::name agent) (time-string (get-universal-time)))
  (static-exit agent "*done*"))

(defun static-test-yell (agent environment)
  "simple skill that prints a message whenever the agent is called"
  (declare (ignore environment))
  (format t "~&YELLING MY NAME: ~A at time ~A"
          (omas::name agent) (time-string (get-universal-time)))
  (static-exit agent "*done*"))
```

We prepare the different scenarios:

```
? (defun goal-say-hello (agent)
    (list (make-instance 'omas::MESSAGE :type :request
                                        :protocol :simple-protocol
                                        :from :TEST :to :TEST
                                        :date (get-universal-time)
                                        :action :PRINT :args nil)))
GOAL-SAY-HELLO

? (defun goal-shout-hello (agent)
    (list (make-instance 'omas::MESSAGE :type :request
                                        :protocol :simple-protocol
                                        :from :TEST :to :TEST
                                        :date (get-universal-time)
                                        :action :SHOUT :args nil)))
GOAL-SHOUT-HELLO
```

```
? (defun goal-yell-hello (agent)
    (list (make-instance 'omas::MESSAGE :type :request
                                        :protocol :simple-protocol
                                        :from :TEST :to :TEST
                                        :date (get-universal-time)
                                        :action :YELL :args nil)))
GOAL-YELL-HELLO
```

We set the different goals:

```
(make-goal :SAY-HELLO :TEST
           :activation-date (+ (get-universal-time) 4)
           :type :cyclic :period 2
           :expiration-date (+ (get-universal-time) 15)
           :script 'goal-say-hello)

(make-goal :SHOUT-HELLO :TEST
           :activation-date (+ (get-universal-time) 5)
           :type :cyclic :period 4
           :expiration-date (+ (get-universal-time) 20)
           :script 'goal-shout-hello)

(make-goal :YELL-HELLO :TEST
           :activation-date (+ (get-universal-time) 2)
           :type :cyclic :period 1
           :expiration-date (+ (get-universal-time) 15)
           :script 'goal-yell-hello)
```

This yields the following trace:

```
YELLING MY NAME: TEST at time 22:37:7
?
YELLING MY NAME: TEST at time 22:37:8
?
Hello, my name is TEST and time is 22:37:9
YELLING MY NAME: TEST at time 22:37:9
?
SHOUTING, my name: TEST at time 22:37:10
YELLING MY NAME: TEST at time 22:37:10
?
Hello, my name is TEST and time is 22:37:11
YELLING MY NAME: TEST at time 22:37:11
?
YELLING MY NAME: TEST at time 22:37:12
?
Hello, my name is TEST and time is 22:37:13
?
YELLING MY NAME: TEST at time 22:37:13
```

```
?
SHOUTING, my name: TEST at time 22:37:14
?
YELLING MY NAME: TEST at time 22:37:14
?
Hello, my name is TEST and time is 22:37:15
?
YELLING MY NAME: TEST at time 22:37:16
?
YELLING MY NAME: TEST at time 22:37:17
?
Hello, my name is TEST and time is 22:37:17
?
YELLING MY NAME: TEST at time 22:37:18
?
SHOUTING, my name: TEST at time 22:37:18
?
YELLING MY NAME: TEST at time 22:37:19
?
Hello, my name is TEST and time is 22:37:19
?
YELLING MY NAME: TEST at time 22:37:20
?
SHOUTING, my name: TEST at time 22:37:22
```

5.3.6 Creating Goals Dynamically

Goals can be created dynamically during the execution of a skill. They can also be removed dynamically, as shown in the previous example, although this is more difficult.

5.4 Memory

The memory mechanism is in reality a set of areas where the designer can store different elements. There is a volatile memory attached to a task, and an agent memory that lasts the time of a session. Then, if one wants to keep information over time, one must use persistency. We will see later that there are other places where one can save data, in particular during the dialogs with a PA.

5.4.1 Memory Attached to a Task

Tasks are executing in a thread. Functions like handlers are executing, then exit. Thus, if one wants to save data in between executions one must use a set of specific functions that save the data by attaching them to the task structure. The needed functions are:

- env-add-values *agent values tag*
- env-get *agent tag*
- env-rem-values *agent values tag &key test*
- env-set *agent values tag*

The functions allow to save a set of values associating them with a tag (preferably a keyword), modify the values by adding or removing some, and recovering them when needed.

This was used in the factorial example of Tutorial 1, for saving the value of nn:

```
;; define a tag (:n) in the environment to record the value of the next
;; products to compute
(env-set agent nn :n)
```

5.4.2 Memory Attached to an Agent

It is possible to save data in the memory of an agent. The data will be saved for the length of the session. This is done with the functions:

- remember *agent fact index*
- recall *agent index*

The remember function allows to save data in a structured memory element: index is usually a keyword. The recall function can be used to retrieve the data.

5.5 Initial State

When starting a short session and making a demo, one sometimes needs to initialize the application by providing data to some of the agents or by predefining messages for debugging the application.

5.5.1 Initializing data

Initializing data in the agent memory can be done with the deffact macro that creates a memory element:

```
(deffact agent fact key)
```

where fact can be any expression and key is usually a keyword.

Initializing data expressions are usually included in the agent definition file.

5.5.2 Predefining Messages

Predefining messages is a convenient way of speeding debugging, since when the application is reloaded the messages need not be redefined and are available in the IDE. Creating messages is done with the defmessage macro:

```
? (defmessage :MSG01 :type :request :to :test :action :hello :args ("Hello!"))
MSG01
? (send-message MSG01)
:MESSAGE-SENT
```

where the fist argument is the name of the message, usually a keyword and the following arguments are the initial arguments needed to specify the message.

Initial messages are usually included in the Z-message file of the application folder.

5.6 Ontology

OMAS ontologies are formalized using the MOSS representation language. See Chapter 12.

5.7 Persistency

Currently persistency is only available for service agents. Of course it is always possible to define skills that use databases. But in the OMAS platform writing

```
(defagent :calendar :persistency t)
```

creates automatically a database partition in which the ontology concepts and knowledge base are saved. The database is opened automatically when starting a new session. Of course while working the skills must read and write objects from and into the database in the habitual sense, i.e. data are commited after changes to be kept.

Persistency is described in details in Chapter 11.

5.8 Display windows

Starting with version 8.1.1 the possibility for a Service Agent to have specific windows has been added. This is far from trivial since an agent may run a number of parallel threads that are more or less transients. Any agent window is created by the scan process, allowing any process executing a skill to access it.

Creating a window and interacting with it is done by using a new performative named :display. An example of dealing with windows can be found in the UTC@DELOS-AUCTION folder in the file defining the DELOS-SELLER (auctioneer).

5.8.1 Creating an SA Window

The window class may be of any type but must be a subtype of OMAS::OMAS-WINDOW. The reason for that is the existence of a special closing method doing some clean up. E.g.

```
(defclass SELLER-WINDOW (OMAS::OMAS-WINDOW)())
```

Creating the window is done from one of the SA skills by sending a message to itself. E.g.

```
(let ((msg (make-instance omas::message :type :display
                          :action :create
                          :from :DELOS-SELLER
                          :to :DELOS-SELLER
                          :args '(:seller-window SELLER-WINDOW
                                    create-seller-window))))
       (send-message msg))
```

The message will be processed by the internal skill :create associated to the :display performative. The arguments are a list of three items:

- the name of the window to be created
- the name of the class of the window
- a function to create the window that will take the two previous parameters as arguments. The function depends on the particular window system.

This creates the window shown Fig.5.1.

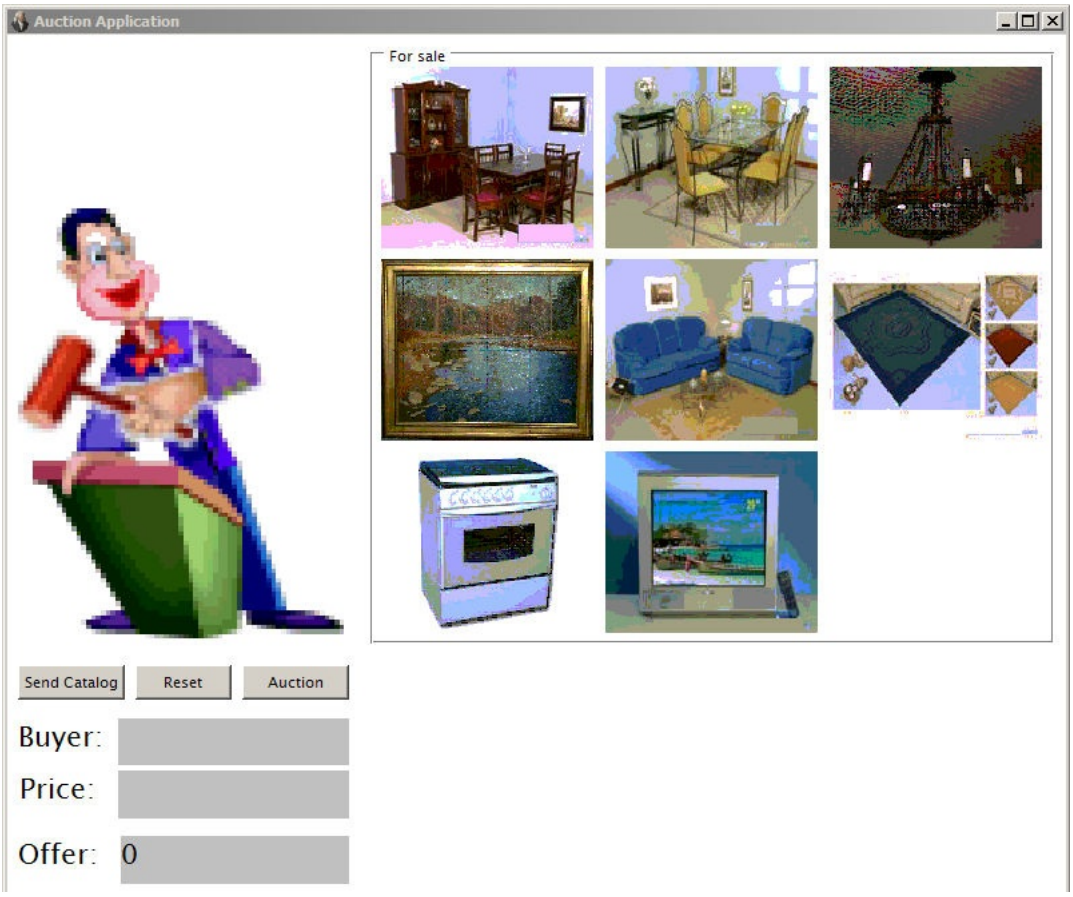

Figure 5.1: DELOS-SELLER window

5.8.2 Interacting with the Window

Callback functions work as usual.

When executing one of the agent skill, interaction may be done directly or by means of messages using the :display performative and one of the following actions: :close, :erase, :execute, :show. This allows other agents to write onto the window (not recommended though).

The only message argument for closing, erasing, or exposing a window is the name of the window. For executing something, two arguments must be provided: the name of the window and an expression to be executed on the window. For example, the following message draws a blue box into the pane of a window named :UML.

```
(omas::message-make
 :type :display
 :to :CONTACT
 :action :execute
 :args '(:UML let ((stream (omas::get-drawable-stream contact::sa_contact
                                                      :uml :uml-draw)))
              (cg:with-background-color (stream cg:blue)
                (cg:draw-box stream (cg:make-box 20 20 50 50))))
)
```

Closing the window by clicking the close button will trigger the appropriate clean up.

5.9 Appendix A - Service Agent Structure

Internally an agent has the following structure, implemented as a set of CLOS objects.

```
=== direct slots
   assistant               ; if T agent is assistant (obsolete, test sub-type)
   gate                    ; sychronizing object
   key                     ; keyword corresponding to its name
   master-if-staff         ; list of masters (keyword) for a staff service agent
   mbox-process            ; process attached to the input-messages mailbox
   name                    ; qualifies the agent (external name) e.g. SA_ADDRESS
   process                 ; process running the agent
   skills                  ; list of skill objects
   traced                  ; t if agent must be traced, nil otherwise

=== sub-objects
appearance
   window                  ; handle to agent (pink) window
   com-window              ; communication channel for a PA
   private-interface       ; user defind interaction window
   thread                  ; unused
   h-pos                   ; horizontal position of lifeline in graphics window
   v-pos                   ; vertical position of lifeline in graphics window
comm
   input-messages          ; list of input-messages
   delayed-input           ; list, used for time-out messages
   output-messages         ; list of output messages
   delayed-output          ; list or results
   displays                ; list of attached windows
   input-log               ; log of all input messages
   output-log              ; log of all input messages
control
   active-task-list        ; list of active tasks (each task in its own thread)
   agenda                  ; list of messages (waiting tasks)
   editing                 ; if true, creating or editing objects
   new-objects             ; when editing saves newly created objects
   pending-bids            ; active bids still pending
                           ; used by the CNet protocol
   processs-list           ; list of processes except for scan, min, and assistant
   saved-answers           ; keeps the unprocessed received answers
   saved-changes           ; we store objects prior to modifying them to be able to
                           ; make an UNDO if needed
   status                  ; :idle
   task-in-progress        ; message corresponding to the task being executed
                           ; unused since each task has its own thread
ontologies
   ontology                ; ? unused
   agent-ontology          ; internal ontology for agent mechanisms
   dialog-ontology         ; ? unused
   domain-ontology         ; e.g. addresses
   language                ; ontology language (e.g. :en :fr or :all)
```

```
   master-ontology          ; ? unused
   moss-context             ; integer referring to MOSS version of the ontologies
   moss-system              ; contains a reference to the value of the special
                            ; *moss-system* special variable (usually $E-SYSTEM.1)
   moss-version-graph       ; version (configuration) graph
   ontology                 ; ? unused
   ontology-file            ; user defined unusual ontology file (e.g. shared file)
   ontology-package         ; agent package (for ontology symbols)
   ontology-window          ; window for displaying ontology
   to-save                  ; contains a list of objects to be saved during the
                            ; next transaction
self
   data                     ; area containing data (user's structured)
   features                 ; features (e.g., :learning)
   goals                    ; long-term goals of the agent
   intentions               ; for implementing BDI agent?
   memory                   ; contains what the agent has learned so far
   moss-ref                 ; handle to MOSS representation of agent
   ontology                 ; ? unused
   persistency              ; if t agent is persistent
tasks
   projects                 ; description of complex tasks (unused yet)
   waiting-tasks            ; description of task being processed (learning agents)
world
   acquaintances            ; other known agents
   environment              ; (obsolete: replaced by data in self sub-object)
   external-services        ; knowledge of other agents
```

5.10 Appendix B - Agent Skill Structure

Skills are CLOS objects

```
   acknowledge-fcn          ; user defined handler to process returned acknowledgments
   bid-cost-fcn             ; user defined function for computing the cost of a bid
   bid-quality-fcn          ; user defined function for computing the quality of a bid
   bid-start-time-fcn       ; user define function for computing earliest start time
   description              ; description of the skill
   dynamic-fcn              ; handler for processing subtask answers
   how-long-fcn             ; user defined function returning time needed to compute
   how-long-left-fcn        ; user defined function returning time left to completion
   name                     ; name of the skill
   preconditions            ; user defined function to chack arguments
   select-best-answer-fcn   ; user defined function to select contract-net answers
   select-bid-fcn           ; user defined function to select contract-net bids
   static-fcn               ; function executed when activating the skill for the
                            ; first time
   static-pattern           ; for checking static arg types
   time-limit-fcn           ; handler for processing time limit interrupts
   timeout-handler          ; handler for processing timeouts
```

Chapter 6

Personal Assistant Agent

Contents

This chapter describes the model of Personal Assistant agent and how to add tasks and associate dialogs for an easy interaction with the master (user).

A Personal Assistant (PA) is a Service Agent interfacing a Human with the platform. Its role is to facilitate interactions. Thus, a PA offers a default interface with its master. A PA is identified to serve one person, to help this person obtain services. The privileged mode of interaction is through a natural language dialog, using keyboard input or vocal input, currently in French, English, Brazilian, Spanish or Japanese. Natural language dialogs are difficult to implement. However, specific mechanisms have been developed to allow constructing such dialogs easily.

Two new features have been added in May 2011:

- the possibility of having a user-defined interaction window
- the possibility of web access

6.1 Creating a Personal Assistant Agent (PA)

Creating a PA can be done with the `defassistant` macro:

```
CG-USER(12): (defassistant :albert)
#<AGENT ALBERT>
```

The defassistant syntax is:

```
(defassistant  <name> {<option> <value>}*)
```

The possibile options are the following:

Option	Usage
:context	specifies the version context for the MOSS objects (default is 0)
:dialog-file	name of a specific dialog file (e.g. shared)
:interface	gives the name of a function in the agent package for building a specific interface window different from the default one, e.g. if true calls MAKE-JEAN-WINDOW.
:language	language for the ontology (may be :all)
:no-window	we do not want to open a PA window
:ontology-file	specifies the pathname of a file containing the ontology to be loaded
:package-nickname	specifies a nickname for the agent package, e.g. :AL
:version-graph	specifies the structure of the different versions of the ontology and the knowledge base (default is version 0)
:voice	if t specifies that the assistant can use a vocal interface
:voice-input-port	PA port of the receive UDP socket (default 52010)
:voice-ip	IP of the machine containing voice component
:voice-output-port	remote port on the voice component (default 52011)

Example (windows XP):

```
(defassistant :albert :language :fr :font "Arial Unicode MS" :size 9)
```

6.1.1 Option language

The language option specifies the language understood by the PA. It must be one of the authorized languages recorded in the moss::*language-tags* parameter.

Using occidental languages like :fr, :en, or :br, does not imply additional changes. However, when adding Japanese language (:jp), then the system must be configured with a Japanese locale and a Japanese input method. Using Japanese requires using a segmentation module different from the occidental segmentation module and currently requires the aclmecab folder to be installed in the same folder as OMAS-MOSS. The font is either the system default font or may be specified as a font accepting Japanese characters with the :font option.

6.1.2 Options font and size

The two options font and size allow specifying the font to be used in the assistant display pane. In the XP or Window 7 environment it must be one of the ACL fonts (see ACL documentation). In the Mac environment it must be one of the MCL fonts (see MCL documentation).

6.1.3 Option show-dialog

The show-dialog option allow to keep track of the exchanges in the dialog. The dialog is printed in the assistant display pane, alternating master's input and assistant answers. By default this option is true.

6.1.4 Option voice

The voice option is used when vocal I/O is performed. The specific arrangements for installing a vocal I/O depend on the vocal recognition system that is used. Detailed in Section 6.20.

6.2 PA Default Interaction Window

After executing the defassistant macro, if the agent ALBERT has indeed been created, there is no visible means of interacting other than sending ALBERT messages like to a standard service agent. The main reason is the necessity to set up a specific interaction machinery, which consists of a set of tasks, an ontology, a dialog and an interaction window. This is done by creating specific files containing the required elements. Such files are then loaded automatically when loading the application (See Chapter 2 Section 2.2.3).

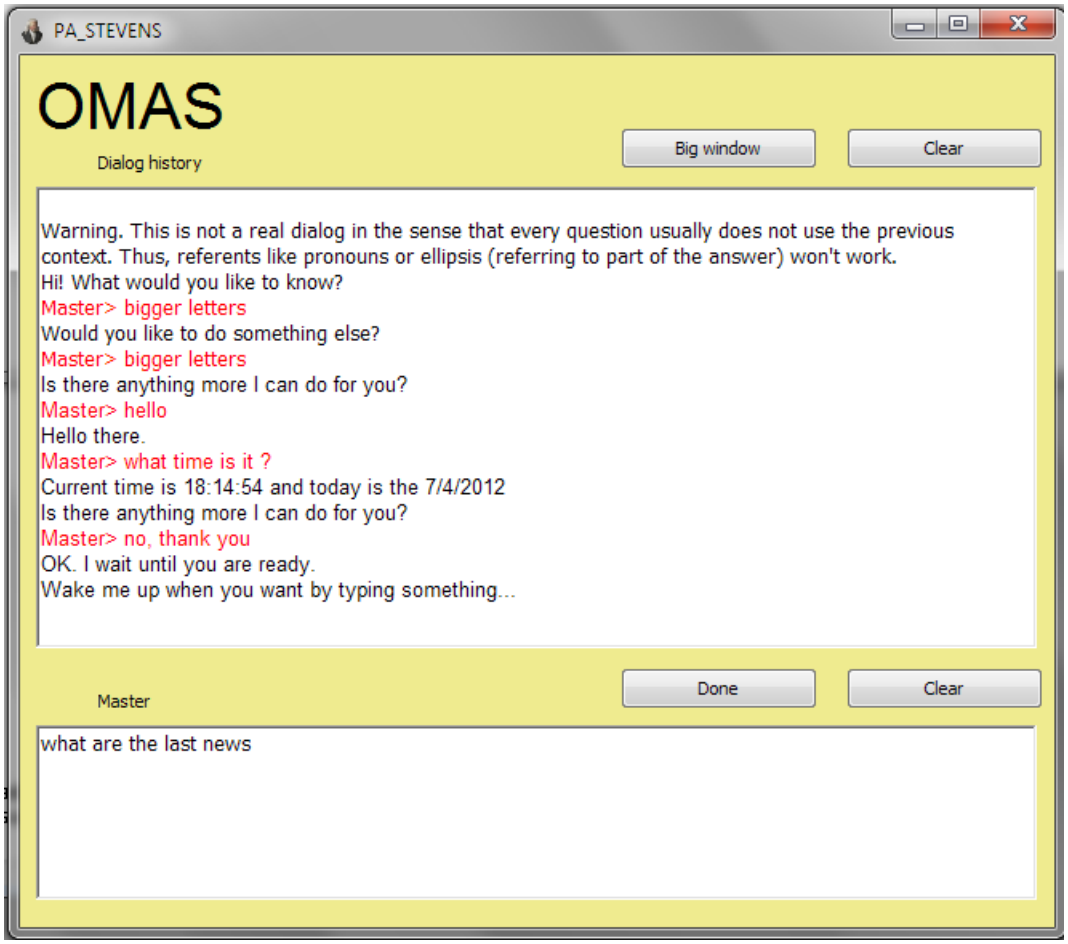

Figure 6.1: OMAS Interaction window for a French Personal Assistant

When loading an agent file, a task file and a dialog file, unless an option has been given to produce

a specific interaction window, a default interaction window appears as shown Fig.6.1. One can then communicate with the PA by typing things in the master's pane of the window.

The Master (user) communicates with her personal assistant through the interface (Fig.6.1). The window has two panes:

- **Dialog history**: an area in which the PA prints questions or answers and displays the history of the dialog. Master data are printed in red ink.
- **Master**: an area in which the master (user) inputs data or requests to the PA.

6.3 Principle of the Dialog with the PA

Dialogs between a user and his PA is organized in such a way as to trigger an action on the PA side. The dialog is composed of a top-level loop in which the PA tries to find out what type of action is requested or if the master simply is giving information. An action results in a task that will be executed after acquiring more information. Each task has an associated conversation, the purpose of which is to obtain the data necessary to execute the task.

Selecting an action (a task) is done as follows:

1. The user tells something to her PA, like "*what are the current projects?*";
2. For each task in the library the PA checks the sentence for phrases specified in the index pattern describing the task, and computes a score by using a MYCIN-like formula[1];
3. Tasks are then ordered by decreasing scores;
4. The task with the higher score is selected as well as all tasks with a score above a specific threshold.
5. The task with the highest score is launched, i.e. its associated dialog is triggered.

Tasks are described in Section 6.4 and dialogs are described in Section 6.5.

6.4 Tasks

6.4.1 The Library of Tasks

Whenever the user asks something, her PA first tries to determine what action is actually meant by the request (or assertion). To do so it uses a library of possible actions, defined as individuals of the TASK concept. Next paragraph shows how the task for obtaining project statistics is defined. The property index pattern gives a list of linguistic cues. Each phrase has a weight between -1 and 1. Note the properties *dialog* and the pair <*where to ask*>, <*message action*>. The first one, *dialog*, is used for calling a dialog to analyze the input and ask for missing information before calling the staff agent(s). It is used when access is done through a PA client interface. The pair <*where to ask*>, <*action*> indicates which staff agent and what skill are concerned with the input, and is used when no dialog is feasible, e.g. when asking questions to the PA using e-mail[2]. It can be safely ignored when using interactive dialogs.

```
(deftask "get-project"
   :doc "Task for getting project statistics"
```

[1] If 2 cues are present, the combined score is computed by the formula a+b-ab

[2] Using e-mail tends to be obsolete and replaced by a web interaction.

```
:dialog _get-project-statistics-conversation
:indexes
   ("project" .4 "projects" .5 "statistics" .4 "stats" .4
    "current" .4 "active" .4 "on going" .4 "actual" .4)
:where-to-ask :PROJECT
:action :statistics-html
)
```

The following points must be made here regarding the choice of the indices and the choice of the weights.

- Indices use terms from the ontology. However, one needs additional terms like "*on going*" or abbreviations that are linguistic additions. Also, one could lemmatize the input sentence, but this has not been found very useful. Selecting the right linguistic cues should be the result of experiments using for example a magician of Oz set up.
- Specifying the weights manually is tricky. The resulting score is used to differentiate among tasks. Thus the weights have to be fine tuned to produce the right answer. An algorithm developed by Gonzalez allows computing the weights using a neural network, but its use is not so easy.
- A threshold value is used to eliminate all tasks with a low score. Default threshold is 0.4. However, when a single task remains in the list of potential tasks, the task is executed, even if its score is under the threshold.

6.5 Dialogs

The DIALOG file is the heart of the interaction mechanism and is by far the most complex. It has three parts:

- a top level conversation;
- a set of task-related sub-conversations (sub-dialogs);
- an escape set of patterns used in case of failure of the analysis of the input text.

6.5.1 The dialog Mechanism

Dialogs are internally represented as automata using the MOSS formalism. A given state of a dialog automaton is a MOSS object. Associated with the state are two methods: =execute and =resume. The automaton is traversed by a *crawler*. When the crawler enters a state it runs the corresponding =execute method. If some information is required from the master or from other agents, then the crawler waits for the answer and then executes the =resume method. In all cases a transition is computed to another state of the automaton, or else a global error is declared and the conversation is restarted.

Constructing the various dialogs is not easy, although some macros are available for simplifying the task. Adapting the dialog to a different language however, can be done by replacing some of the strings without worrying too much about the mechanism.

We examine first the top-level conversation (dialog), then an example of a sub-conversation (sub-dialog). The corresponding code can be found in the examples of applications in the OMAS/applications folder.

6.5.2 Top-Level Conversation

The Top-Level conversation is used to select a task from the list of tasks. It can be modified. However, **I do not recommend to modify it** other than translating the various strings into the target language counterparts.

The top-level conversation loop is shown Fig.6.2. The graph shows an entry state, a sub-dialog with a return onto the "More?" state, and three transitions possible to a "Sleep" state, back to the "Main Conversation Process" sub-dialog, or to the "Get input" state. The "Dialog Abort" state corresponds to a reset of the conversation when an error occurs and no transition can be found.

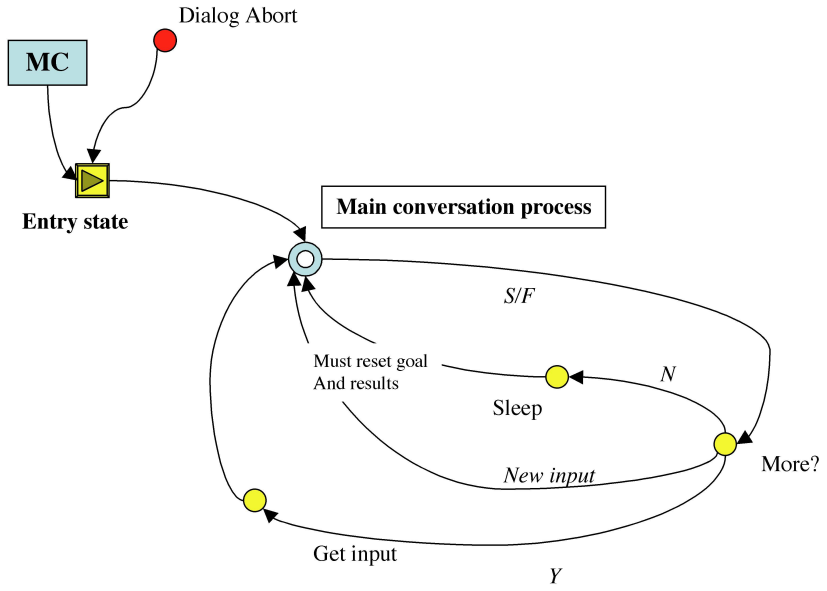

Figure 6.2: PA Main Conversation, showing the "Main Conversation Process" sub-dialog

Let us now examine step by step how the dialog proceeds.

Top-Level Dialog Entry State

The entry state structure is composed of a Dialog-Header individual, referenced by the global _main-conversation variable (ALBERT-DIALOG.lisp file). All the dialog structures are created in the ALBERT name space (ALBERT package), meaning that each PA has its own dialog structures independent of any other agent.

```
;;;=============================================================================
;;;
;;;                  Conversations and Sub-Conversations
;;;
;;;=============================================================================

;;; the following conversations are defined thereafter
;;; the entry point of the dialog is _MAIN-CONVERSATION

(eval-when (:load-toplevel :compile-toplevel :execute)
  (proclaim '(special
              _MAIN-CONVERSATION
```

```
              ...
              )))

;;;============================================================================
;;;
;;;                         MAIN CONVERSATION (CONTROL LOOP)
;;;
;;;============================================================================

;;; create a dialog header to be used for the main conversation

(defindividual
  MOSS-DIALOG-HEADER
  (HAS-MOSS-LABEL "Main conversation loop")
  (HAS-MOSS-EXPLANATION "This is the main conversation loop.")
  (:var _main-conversation))
```

The main conversation is linked once and for all to the agent by inserting a pointer into the agent structure:

```
;; declare that conversation, replacing default conversation
(setf (omas::dialog-header PA_ALBERT) _main-conversation)
```

The states of the main conversation are declared as global variables:

```
;;;===== states are declared as global variables
;;; the set of states can be considered as a plan to be executed for conducting the
;;; conversation with the master

;;;----------------------------------------------------------------------------
(eval-when (:compile-toplevel :load-toplevel :execute)
  (proclaim '(special
              _mc-entry-state
              _mc-get-input
              _mc-more?
              _mc-process
              _mc-sleep)))
;;;----------------------------------------------------------------------------
```

The entry-state structure can now be defined:

```
(defstate _mc-entry-state
  (:entry-state _main-conversation)
  (:label "Début du dialogue")
  (:explanation
   "Initial state when the assistant starts a conversation. Send a welcome message ~
    and wait for data. Also entered on a restart following an abort.")
  (:reset-conversation)
  (:text "Attention! Ce dialogue est trés limité. Chaque phrase est indépendante ~
          et l'on ne peut utiliser des pronoms référents.~%")
  (:question-no-erase
```

```
 ("- Bonjour ! que puis-je faire pour vous ?"
   "- Salut ! Que voudriez-vous savoir ?"
   "- Bonjour ! Je vais faire de mon mieux pour répondre à vos ~
    questions. Mais, rappelez-vous que mon QI est faible."))
(:transitions
 (:always :target _mc-process))
)
```

It reads as follows:

- the defstate macro creates the node structure and the associated methods;
- the global variable pointing to the state is _main-conversation declared to be an entry-state;
- a label indicates the name of the state "Début du dialogue";
- an explanation provides some documentation;
- the :reset-conversation option reset all the internal variables of the dialog;
- the :text option prints the associated text;
- the :question-no-erase option prints the associated text selecting randomly one of the following strings (the number of strings or choices is not limited);
- the :transitions option has sub-options:
  - :always means that the transition will always be executed;
  - :target identifies the transition state.

On the screen the process starts as follows (Fig.6.3):

- a welcome message is printed into the assistant pane;
- the crawler waits for the master to type or to say something (vocal interface).

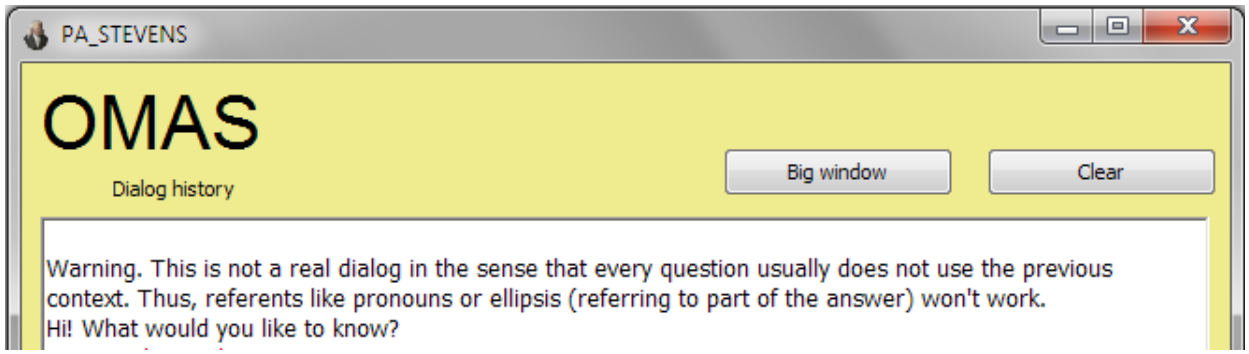

Figure 6.3: Initial opening of the Main Conversation dialog

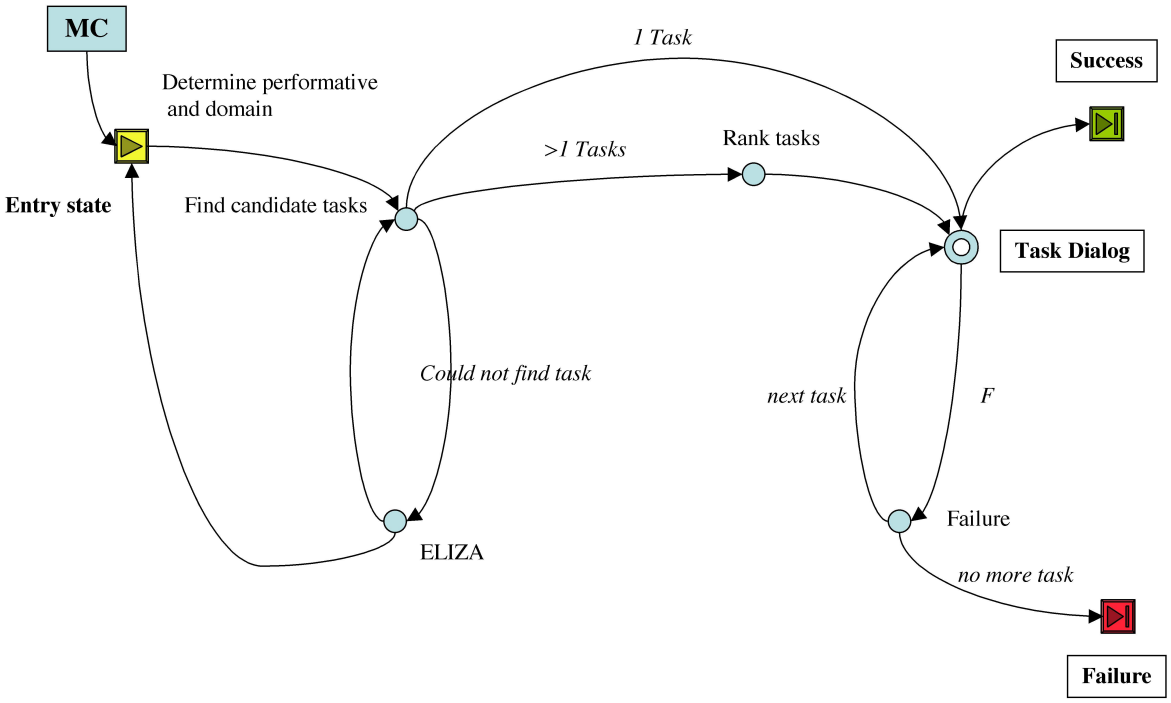

Figure 6.4: Main Conversation Process sub-dialog

Main Conversation Process Sub-Dialog

After the master's input the crawler makes a transition to the "Main Conversation Process" sub-dialog that can be seen as a subroutine in a traditional programming language. The corresponding automaton is shown Fig.6.4.

The sub-dialog includes an entry-state, and two exit states respectively labeled "Success" and "Failure." It also includes a new sub-dialog called "Task Dialog."

The statements defining the sub-dialog in the ALBERT-DIALOG.lisp file are the following:

```
;;;=============================================================================
;;;
;;;                        MAIN CONVERSATION (EXECUTION PART)
;;;
;;;=============================================================================

;;; this conversation is intended to process the input from te user by first
;;; determining the performative, then the list of possible tasks

(defsubdialog
  _process-conversation
  (:label "Process conversation")
  (:explanation
   "Processing steps of the main conversation.")
  (:states _mc-eliza
           _mc-find-performative
           _mc-entry-state
           _mc-failure
```

```
          _mc-find-performative
          _mc-find-task
          _mc-select-task
          _mc-task-dialog)
  )
```

Main Conversation Process Entry State

The entry state of the dialog is in fact the "Find Performative" state defined as follows:

```
;;;------------------------------------------------------- (MC) FIND-PERFORMATIVE

(defstate _mc-find-performative
  (:entry-state _process-conversation)
  (:process-state _main-conversation)
  (:label "Find performative")
  (:explanation
   "Process what the user said trying to determine the type of performative ~
    among :request :assert :command. Put the result if any into the performative ~
    slot of the conversation object.")
  (:transitions
   ;; time?
   (:patterns (("Quelle" "heure" (?* ?x)))
              :exec (moss::print-time moss::conversation :fr) :success)
   ;; now really select performatives
   (:patterns (("qui" *)
               ("quel" "est" *)
               ("quelle" "est" *)
               ("quels" "sont" *)
               ("quelles" "sont" *)
               ("qu" "est-ce" "que" *)
               ("quoi" *)
               ("quand" *)
               ("où" *)
               ("pourquoi" *)
               ("qui" *)
               ("combien" *)
               ("comment" *)
               ("est-ce" "que" *)
               ("est-il" *)
               ("est" "il" *)    ; interface vocal ?
               ("est-elle" *)
               ("est" "elle" *)
               ("sont-ils" *)
               ("sont" "ils" *)
               ("sont-elles" *)
               ("sont" "elles" *)
               ("a-t-il" *)
               ("a-t-elle" *)
               ("ont-ils" *)
```

```
              ("ont-elles" *)
              (* "c" "est" "quoi" *)
              (* "c" "est" "qui" *)
              (* "?")
              )
             :set-performative (list :request)
             :target _mc-find-task)
  (:patterns ((*"notez" *)
              (* "noter" *))
             :set-performative (list :assert)
             :target _mc-find-task)
  (:otherwise
   :set-performative (list :command)
   :target _mc-find-task))
)
```

The macro reads as follows:

- the :entry-state option indicates that the state is the entry state of the sub-dialog;
- the :process-state option is currently unused;
- the :label and :explanation options are self explanatory;
- the :transitions option computes a possible transition from the saved input contained in the FACTS/INPUT area of the conversation object. It has sub-options:
  - the first :patterns option tests whether the master is asking for the time by checking if the input starts with "Quelle heure" (What time), in which case the

    ```
    (moss::print-time moss::conversation :fr)
    ```

    is executed and the sub-dialog returns with a success.
  - the second :patterns option checks whether the input starts or contains different words or phrases that will qualify it to be a question (request performative). If so, it will set the performative (:set-performative), and make a transition to the "Find Task" state (:target);
  - the third :patterns option checks if the input contains "notez" or "noter" that will qualify the input to be an assertion (assert performative). If so, it will set the performative (:set-performative), and make a transition to the "Find Task" state (:target);
  - the :otherwise option sets the performative to :command, and make a transition to the "Find Task" state (:target);

The defstate macro produces the state structure and the associated methods =execute and =resume. The code is sometimes difficult to visualize. However, it is possible to see the code by using a debug mode (See Section 6.7.1).

Main Conversation Process Find Task State

The "Find Task" state is somewhat more complex because the defstate macro is not powerful enough to synthesize the =execute method. Consequently we must write a special =answer-analysis method by hand as follows:

```
;;;------------------------------------------------------------------ (MC) FIND-TASK

(defstate _mc-find-task
  (:label "Find task")
  (:explanation
   "Process the input to determine the task to be undertaken. Combines the words ~
    from the sentence to see if they are entry points for the index property of ~
    any task. Collect all tasks for which there is an index in the sentence.")
  (:answer-analysis)
  )

(defownmethod
  =answer-analysis _mc-find-task (conversation input)
  "Using input to find tasks If none failure, if one, OK, if more, must ask user to ~
   select one. Checks for an entry point for an index of a task.
Arguments:
   conversation: current conversation
   input: list containing the input as a list of words"
  (let* ((performative (read-fact conversation :PERFORMATIVE))
         task-list)
    (moss::vformat "_mc-find-task /performative: ~S, package: ~S, input:~%  ~S"
                                performative *package* input)
    ;; try to find a task from the input text
    (setq task-list (moss::find-objects
                     '(TASK
                            (HAS-INDEX-PATTERN
                            (TASK-INDEX (HAS-INDEX :is :?))))
                     input :all-objects t))
(moss::vformat "_mc-find-task /possible tasks:~%  ~S" task-list)
    ;; filter tasks that do not have the right performative
    (setq task-list
          (mapcan #'(lambda (xx)
                      (if (intersection performative (HAS-PERFORMATIVE xx))
                        (list xx)))
                  task-list))
    (moss::vformat "_mc-find-task /task list after performative check: ~S" task-list)
    (cond
     ;; if empty, ask ELIZA
     ((null task-list)
      `(:transition ,_mc-ELIZA))
     ;; if one, then OK
     ((null (cdr task-list))
      ;; save results, make the task the conversation task
      (setf (HAS-MOSS-TASK conversation) task-list)
      `(:transition ,_mc-task-dialog))
     (t
      ;; otherwise must select one from the results
      (setf (HAS-MOSS-TASK-LIST conversation) task-list)
      `(:transition ,_mc-select-task)))
    ))
```

Let us examine the manual =answer-analysis method.

```
(let* ((performative (read-fact conversation :PERFORMATIVE))
       task-list)
```

First we recover the parformative from the FACTS area of the conversation object. A local task-list variable is declared.

Then, the MOSS find-objects function is called with the list of words contained in the input variable to extract from the ALBERT knowledge base the tasks that contain some of the words as indices (task-index).

```
;; try to find a task from the input text
(setq task-list (moss::find-objects
                 '(task (has-index-pattern (task-index (has-index :is :?))))
                 input :all-objects t))
```

The result may be NIL, contain one task, or contain several tasks.

Then, the task are filtered according to the type of performative:

```
;; filter tasks that do not have the right performative
(setq task-list
      (mapcan #'(lambda (xx)
                  (if (member performative (HAS-PERFORMATIVE xx))
                    (list xx)))
              task-list))
```

Again, the result may be NIL, contain one task, or contain several tasks.

Then a transition is selected according to the result:

```
(cond
 ;; if empty, ask ELIZA
 ((null task-list)
  `(:transition ,_mc-ELIZA))
 ;; if one, then OK
 ((null (cdr task-list))
  ;; save results, make the task the conversation task
  (setf (HAS-MOSS-TASK conversation) task-list)
  `(:transition ,_mc-task-dialog))
 (t
  ;; otherwise must select one from the results
  (setf (HAS-MOSS-TASK-LIST conversation) task-list)
  `(:transition ,_mc-select-task)))
```

If no task are left we have a failure (i.e. we did not understood what the master said), in which case we make a transition to the ELIZA state. If we have a single task left, we execute it. If we have more than one task, we must select one and make a transition to the "Select Task" state. In all cases the =answer-analysis method returns the following pattern:

```
(:transition <internal ID of a state>)
```

The moss::vformat function is used for debugging.

Main Conversation Process Select Task State

The role of the "Select Task" state is to select one of the tasks to execute. IAgain, the defstate macro is not powerful enough to synthesize the =execute method and we must provide an =answer-analysis method:

```
;;;------------------------------------------------------------ (MC) SELECT-TASK

(defstate
  _mc-select-task
  (:label "mc select task")
  (:explanation "we have located more than one task. We rank the tasks by computing ~
                 the average weight of the terms in the task index slot. We then ~
                 record the list of tasks into the statte-context task-list slot ~
                 and activate the highest ranking task.")
  (:answer-analysis)
  )

(defownmethod =answer-analysis _mc-select-task (conversation input)
  "we have located more than one task. We rank the tasks by computing ~
   the MYCIN combination weight of the terms in the task index slot. We then ~
   record the list of tasks into the conversation task-list slot ~
   and activate the highest ranking task."
  ;(declare (ignore input))
  ;; the list of tasks is in the HAS-MOSS-TASK-LIST slot. Tasks are defined in the
  ;; PA-tasks.lisp file in the application folder
  (let* ((task-list (HAS-MOSS-TASK-LIST conversation))
         patterns weights result pair-list selected-task level word-weight-list)
    (moss::vformat "_select-task /input: ~S~&  task-list: ~S" input task-list)

    ;; first compute a list of patterns (combinations of words) from the input
    (setq patterns (mapcar #'car (moss::generate-access-patterns input)))
    (moss::vformat "_select-task /input: ~S~&  generated patterns:~&~S"
                   input patterns)

    ;; then, for each task
    (dolist (task task-list)
      (setq level 0)
      ;; get the weight list
      ;(setq weights (HAS-INDEX-WEIGHTS task))
      (setq weights (moss::%get-INDEX-WEIGHTS task))
      (moss::vformat "_select-task /task: ~S~&  weights: ~S" task weights)
      ;; check the patterns according to the weight list
      (setq word-weight-list (moss::%get-relevant-weights weights patterns))
      (moss::vformat "_select-task /word-weight-list:~&  ~S" word-weight-list)
      ;; combine the weights
      (dolist (item word-weight-list)
        (setq level (+ level (cadr item) (- (* level (cadr item))))))
      (moss::vformat "_select-task /level: ~S" level)
      ;; push the task and weight onto the result list
      (push (list task level) result)
```

```
    )
  (moss::vformat "_select-task /result:~&~S" result)

  ;; order the list
  (setq pair-list (sort result #'> :key #'cadr))
  (moss::vformat "_select-task /pair-list:~&  ~S" pair-list)
  ;; keep the first task whatever its score
  (setq selected-task (caar pair-list))

  ;; remove the task that have a weight less than task-threshold (default 0.4)
  (setq pair-list
        (remove nil
                (mapcar
                 #'(lambda (xx)
                     (if (>= (cadr xx) (omas::task-threshold omas::*omas*)) xx))
                 pair-list)))
  ;; if task-list is empty then return the first saved task
  ;; this may not be a good policy if the score is too low
  (if (null pair-list)
    (progn
      ;; reset the task-list slot of the conversation object
      (setf (HAS-MOSS-TASK-LIST conversation) nil)
      ;; put the saved task into the task slot
      (setf (HAS-MOSS-TASK conversation) (list selected-task))
      ;; go to task-dialog
      `(:transition ,_mc-task-dialog)
      )
    (progn
      ;; remove the weights
      (setq task-list (mapcar #'car pair-list))
      ;; select the first task of the list
      (setq selected-task (pop task-list))
      (moss::vformat "_select-task /selected task: ~S" selected-task)
      ;; save the popped list in the task-list slot of the conversation object
      (setf (HAS-MOSS-TASK-LIST conversation) task-list)
      (setf (HAS-MOSS-TASK conversation) (list selected-task))
      ;; go to task-dialog
      `(:transition ,_mc-task-dialog)))
  ))
```

The first part is similar to the previous case, recovering the list of tasks from the conversation object and declaring local variables:

```
(let* ((task-list (HAS-MOSS-TASK-LIST conversation))
       patterns weights result pair-list selected-task level word-weight-list)
```

Then, using the MOSS internal generate-access-patterns, the words contained in the input are combined to produce patterns that will be checked against the index patterns of each task:

```
  ;; first compute a list of patterns (combinations of words) from the input
  (setq patterns (mapcar #'car (moss::generate-access-patterns input)))
```

Then, for each task a score is computed using the MYCIN combination and a list of sublists (task score) is produced (result variable):

```
;; then, for each task
(dolist (task task-list)
  (setq level 0)
  ;; get the weight list
  ;(setq weights (HAS-INDEX-WEIGHTS task))
  (setq weights (moss::%get-INDEX-WEIGHTS task))
  (moss::vformat "_select-task /task: ~S~&  weights: ~S" task weights)
  ;; check the patterns according to the weight list
  (setq word-weight-list (moss::%get-relevant-weights weights patterns))
  (moss::vformat "_select-task /word-weight-list:~&  ~S" word-weight-list)
  ;; combine the weights
  (dolist (item word-weight-list)
    (setq level (+ level (cadr item) (- (* level (cadr item))))))
  (moss::vformat "_select-task /level: ~S" level)
  ;; push the task and weight onto the result list
  (push (list task level) result)
  )
```

Then, the list contained in result is ordered by decreasing weights:

```
;; order the list
(setq pair-list (sort result #'> :key #'cadr))
```

Then save the first task:

```
;; keep the first task whatever its score
(setq selected-task (caar pair-list))
```

Then we remove the tasks that are under a given threshold:

```
(setq pair-list
      (remove nil
              (mapcar
               #'(lambda (xx)
                   (if (>= (cadr xx) (omas::task-threshold omas::*omas*)) xx))
               pair-list)))
```

Then, if there is nothing left we execute the saved task, otherwise we select the highest task and keep the rest in case of failure.

```
(if (null pair-list)
  (progn
    ;; reset the task-list slot of the conversation object
    (setf (HAS-MOSS-TASK-LIST conversation) nil)
    ;; put the saved task into the task slot
    (setf (HAS-MOSS-TASK conversation) (list selected-task))
    ;; go to task-dialog
    `(:transition ,_mc-task-dialog)
    )
  (progn
```

```
        ;; remove the weights
        (setq task-list (mapcar #'car pair-list))
        ;; select the first task of the list
        (setq selected-task (pop task-list))
        (moss::vformat "_select-task /selected task: ~S" selected-task)
        ;; save the popped list in the task-list slot of the conversation object
        (setf (HAS-MOSS-TASK-LIST conversation) task-list)
        (setf (HAS-MOSS-TASK conversation) (list selected-task))
        ;; go to task-dialog
        `(:transition ,_mc-task-dialog)))
```

Again, the moss::vformat function is used for debugging.

Main Conversation Process Task Dialog State

The Main Conversation Process Task Dialog state calls the sub-dialog associated with the task to execute. An example will be detailed in Section 6.6. The state code is the following:

```
;;;------------------------------------------------------------- (MC) TASK-DIALOG

(defstate _mc-task-dialog
  (:label "Task dialog")
  (:explanation
   "We found a task to execute (in the GOAL slot of the conversation). We activate ~
    the dialog associated with this task.")
  ;; we launch the task dialog as a sub-conversation
  ;; the task contains the name of a sub-conversation header, e.g. _get-tel-nb
  (:answer-analysis)
  )

(defownmethod
  =answer-analysis _mc-task-dialog (conversation input)
  "We prepare the set up to launch the task dialog.
Arguments:
   conversation: current conversation"
  (declare (ignore input))
  (let* ((task-id (car (send conversation '=get 'HAS-MOSS-TASK))))
    (moss::vformat "_mc-task-dialog /task-id: ~S dialog: ~S"
                   task-id (HAS-DIALOG task-id))
    ;; we launch the task dialog as a sub-conversation
    ;; the task contains the name of a sub-conversation header, e.g. _get-tel-nb
    `(:sub-dialog ,(car (HAS-DIALOG task-id)) :failure ,_mc-failure)
    ))
```

The =answer-analysis method returns the pattern:

```
   (:sub-dialog <sub-conversation header>)
                :failure <state internal ID for transition on failure>)
```

Main Conversation Process Failure State

A transition to this state occurs when the executed task returns with a failure. The current policy is to execute the next task from the list of tasks if any is left, otherwise to return the :failure pattern:

```
;;;------------------------------------------------------------------ (MC) FAILURE

(defstate _mc-failure
  (:label "Failure state of main conversation")
  (:explanation
   "Failure state is entered when we return from a sub-dialog with a :failure tag. ~
    We check for more tasks to perform (listed in the task-list slot of the ~
    conversation object). If there are more, we execute the first one. If there ~
    are no more, we return with a failure tag.")
  (:answer-analysis)
  )

(defownmethod
  =answer-analysis _mc-failure (conversation input)
  (declare (ignore input))
  (let* ((task-list (HAS-MOSS-TASK-LIST conversation))
         task)
    (if task-list
      (progn
        ;; record next task
        (setq task (pop task-list))
        ;; remove it from task-list
        (send conversation '=replace 'HAS-MOSS-TASK-LIST task-list)
        ;; add it to conversation
        (send conversation '=replace 'HAS-MOSS-TASK (list task))
        ;; transfer to sub-dialog
        `(:transition ,_mc-task-dialog))
      ;; when no more tasks we return :failure and the state variable still contains
      ;; :failure. The crawler will return one more level until hitting a non
      ;; failure return containing a specific state (e.g. the one for the main
      ;; conversation)
      (list :failure)
      )))
```

The code is self explanatory.

Main Conversation Eliza State

The ELIZA state is a special state that is visited when the PA cannot make any sense of the master's input. It is intended to trigger small talk rather than letting the PA answer "I did not understand, please rephrase your request..." ELIZA has been copied from Peter Norvig's book on Artificial Intelligence programming. The code is the following:

```
;;;------------------------------------------------------------------ (MC) ELIZA

(moss::defstate _mc-ELIZA
  (:label "ELIZA")
  (:explanation
   "Whenever MOSS cannot interpret what the user is saying, ELIZA is called to ~
    do some meaningless conversation to keep the user happy. It then record the ~
```

```
    master's input and transfers to find performative state.")
  (:eliza)
  (:transitions
   (:always :target _mc-find-performative))
  )
```

The :eliza option triggers the ELIZA sub-dialog.

6.6 Task Subdialog

Task dialogs may be simple or very complex, depending on the task to be executed. We examine two different dialogs found in the ALBERT-DIALOG.lisp file. For each task to be executed by the agent, an associated dialog must be provided.

6.6.1 The Print Help Sub-Dialog

This is an example of very simple sub-dialog associated with the Print Help task. It has only two states declared as follows:

```
;;;============================================================================
;;;
;;;                        PRINT HELP CONVERSATION
;;;
;;;============================================================================

;;; this conversation is intended to help the master by giving information:
;;;   - in general (what tasks are available
;;;      "what can you do for me?" "help." "what can I do?"
;;;   - on a particular topic
;;;      "how do I send a mail?" "help me with the mail?"
;;; specific help corresponding to special help tasks are caught by the task
;;; task selection mechanism?

(defsubdialog
  _print-help-conversation
  (:label "Help conversation")
  (:explanation
   "Help was asked.")
  (:states _ph-print-global-help)
  )
```

The corresponding automaton is shown Fig.6.5.

Print Help Entry State

The Print Help entry state simply prints the information and returns a success.

```
;;;------------------------------------------------------ (PH) PRINT-GLOBAL-HELP

(defstate
  _ph-print-global-help
  (:label "Print general help")
```

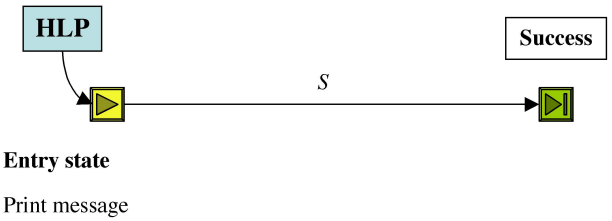

Figure 6.5: Print Help sub-dialog

```
(:entry-state _print-help-conversation )
(:explanation "No specific subject was included.")
(:execute
 (let
   ((obj-id (car (send '>-global-help '=get 'is-title-of)))
    (*language* :fr)
    (task-list (access '(task)))
    (conversation (omas::conversation HDSRI::pa_HDSRI))
    )
   (when obj-id (send obj-id '=get-documentation)
         (send conversation '=display-text *answer*))
   (send conversation '=display-text
         "~2%Je peux faire les choses suivantes :")
   (dolist (task task-list)
     (send task '=get-documentation :lead "   - ")
     (send conversation '=display-text *answer*)
     )
   (send conversation '=display-text "~2%")
   ))
(:transitions
 (:always :success))
)
```

6.6.2 The Get Address Sub-Dialog

The Get Address sub-dialog is more complex and uses a staff agent to complete the required task. It has the following states:

```
;;;=============================================================================
;;;
;;;                       GET ADDRESS CONVERSATION
;;;
;;;=============================================================================

;;; this conversation is intended to print the address of a person or of a
;;; place. It fills the task pattern and sends the message to the address specialist.
;;; The simplest version is to send to the :SA-ADDRESS agent, wait for the result and
;;; print it.
```

```
(defsubdialog
  _get-address-conversation
  (:label "Get address conversation")
  (:states _get-address-dialog  ; required by defstate
           _gad-entry-state
           _gad-dont-understand
           _gad-try-again
           _gad-sorry)
  (:explanation
   "Master is trying to obtain an address.")
  )
```

This corresponds to the automaton shown Fig.6.6.

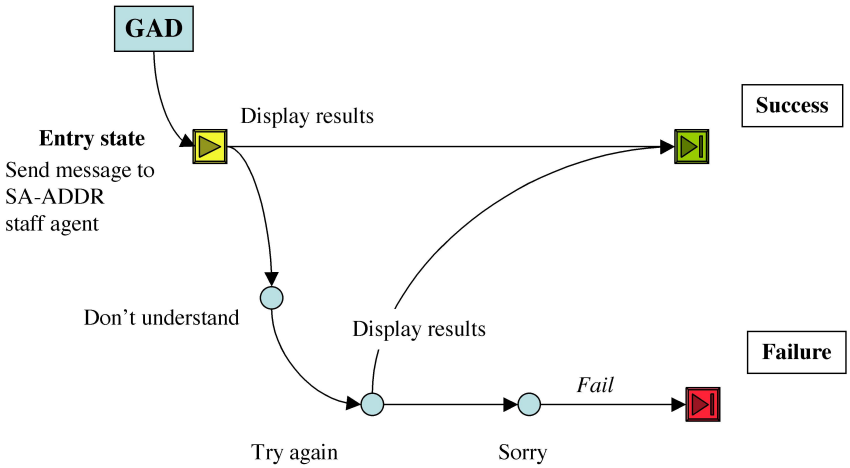

Figure 6.6: Get Address sub-dialog

Get Address Entry State

The code is as follows:

```
;;;------------------------------------------------ (GAD) GET-ADDRESS-ENTRY-STATE

(defstate
  _get-address-entry-state
  (:entry-state _get-address-conversation)
  (:label "Get address dialog entry")
  (:explanation "Assistant is sending a free-style message to the ADDRESS agent.")
  (:send-message :to :ADDRESS :self PA_ALBERT :action :get-address
                 :args `(((:data . ,(moss::read-fact moss::conversation :input))
                          (:language . :fr)))
                 )
  (:transitions
   (:on-failure :target _gad-dont-understand)
   (:test (> (length (read-fact moss::conversation :answer)) 3)
          :exec
```

```
          (progn
            ;; record the pattern for printing selection summaries into the
            ;; working list
            (replace-fact moss::conversation :control-string "~{~A~^ ~}, ~{~A~^ ~}")
            (replace-fact moss::conversation :properties (list "nom" "prénom"))
            )
          :sub-dialog _make-choice-conversation :failure _gad-dont-understand)
   (:otherwise
    :print-answer #'print-addresses :success))
  )
```

The :send-message option sends a message to the :ADDRESS staff agent with the request :get-address and the argument composed of a single list:

```
`(((:data . ,(moss::read-fact moss::conversation :input))
    (:language . :fr)))
```

Data includes the input from the user (to be found in the FACTS/INPUT area of the conversation object), and language specifies the language to be used because staff agents can have multilingual capacities. Note that the conversation object is in the MOSS package.

If the answer from the staff agent is a failure, we transfer to the dont-understand state. Otherwise, if we have more than 3 answers, we use the FACT area of the conversation object to store a control string and the names of the properties we want to print as a summary and transfer to the make-choice subdialog. If we have less than 3 answers, we print them and return a success.

Dont Understand and Sorry States

The "Dont Understand and Sorry states are self explanatory:

```
;;;------------------------------------------------------ (GAD) GAD-DONT-UNDERSTAND

(defstate
  _gad-dont-understand
  (:label "Did not understand. Ask master.")
  (:explanation "could not find the address with the given information. Thus, ~
                 asking master for whose address...")
  (:question-no-erase "~%- L'adresse de qui ?")
  (:answer-type :answer)
  (:transitions
   (:always :target _gad-try-again)))

;;;------------------------------------------------------------------ (GAD) GAD-SORRY

(defstate
  _gad-sorry
  (:label "Address failure.")
  (:explanation "We don't have the requested address.")
  (:text "Désolé, si vous recherchiez une adresse, je ne la trouve pas.~%")
  (:reset)
  (:transitions (:failure))
 )
```

Try Again State

The Try again sends a message again to the staff agent with the master's answer from the "don't understand" state:

```
;;;------------------------------------------------------------ (GAD) GAD-TRY-AGAIN

(defstate
  _gad-try-again
  (:label "Ask again for address.")
  (:explanation "Assistant is asking :ADDRESS.")
  (:send-message :to :ADDRESS :self PA_ALBERT :action :get-address
                 :args `(((:data . ,(read-fact moss::conversation :input))
                          (:language . :fr))))
  (:transitions
   (:on-failure :target _gad-sorry)
   (:otherwise
    :print-answer #'print-address
    :success))
  )
```

This is similar to the entry state code except for the transitions.

Make Choice Sub-conversation

The Make Choice sub-dialog is not detailed here.

6.7 System Internals

Debugging is difficult and can be somewhat improved as explained in the next sections.

6.7.1 Viewing the defstate Code

The defstate macro produces code which may be examined as follows:

- set the moss::*debug* global variable to T, which can be done by calling (d+);
- execute the (defstate ...) expression.
  Example: if we execute the following expression:

  ```
  (defstate _mc-SLEEP
    (:label "Nothing to do")
    (:explanation
     "User said she wanted nothing more.")
    (:reset)
    (:text "- OK. J'attends que vous soyez prêt.~%")
    (:question-no-erase
     "- Réveillez moi en tapant ou en disant quelque chose...")
    (:transitions
     (:always :target _mc-process)))
  ```

  OMAS will print:

```
;***** Debug: STATE: _MC-SLEEP

;*** Debug: state: _MC-SLEEP =EXECUTE
   (MOSS::%MAKE-OWNMETHOD
     =EXECUTE
     _MC-SLEEP
     (MOSS::CONVERSATION &REST MOSS::MORE-ARGS)
     "*see explanation attribute of the _MC-SLEEP state*"
     (DECLARE (IGNORE MOSS::MORE-ARGS))
     (SEND MOSS::CONVERSATION
           '=DISPLAY-TEXT
           '"- OK. J'attends que vous soyez prêt.~%"
           :ERASE
           T)
     (SEND MOSS::CONVERSATION
           '=DISPLAY-TEXT
           '"- Réveillez moi en tapant ou en disant quelque chose...")
     '(:WAIT))

;*** Debug: state: _MC-SLEEP =RESUME
   (MOSS::%MAKE-OWNMETHOD
     =RESUME
     _MC-SLEEP
     (MOSS::CONVERSATION)
     "*see explanation attribute of the _MC-SLEEP state*"
     (LET* (MOSS::RETURN-VALUE)
       (CATCH :RETURN
         (COND ((INTERSECTION MOSS::*ABORT-COMMANDS*
                              (READ-FACT MOSS::CONVERSATION :INPUT)
                              :TEST
                              #'STRING-EQUAL)
                (THROW :DIALOG-ERROR NIL))
               (T (SETQ MOSS::RETURN-VALUE (LIST :TRANSITION _MC-PROCESS)))))
       MOSS::RETURN-VALUE))
_MC-SLEEP
```

In this mode states are not loaded, but simply displayed. To load a state, e.g. after a correction, one must return to normal execution by resetting the moss::*debug* variable to NIL, which can be done by calling the (d-) function.

6.7.2 The MOSS vformat Macro

vformat is a macro defined in the :moss package as follows:

```
(defMacro vformat (cstring &rest args)
  `(if *verbose* (format *debug-io* ,(concatenate 'string "~&;***** " cstring)
                         ,@args)))
```

Thus, vformat prints to the *debug-io* channel whenever moss::*verbose* is not nil.

6.7.3 Tracing the Dialog

Tracing the dialog is obtained by setting the following global variables:

```
(setq moss::*transition-verbose* t) or (v+)
(setq moss::*traced-agent* albert::ALBERT)
```

6.8 More on the **defstate** Macro

defstate is a key macro for writing dialogs. I present several examples of its use in this paragraph.

6.8.1 A Simple Use

When we simply want to ask a question and do a simple processing on the result, then the syntax is easy. Example from the Main Conversation More? state:

```
(defstate _mc-more?
  (:label "More?")
  (:explanation "Asking the user it he wants to do more interaction.")
  (:reset-conversation)
  (:question-no-erase
   ("Que puis-je faire d'autre pour vous ?"
    "Y a-t-il autre chose que je puisse faire pour vous ?"
    "Avez-vous d'autres questions ?"
    "OK.")
   )
  (:transitions
   (:starts-with ("rien") :target _mc-sleep)
   (:no :target _mc-sleep)
   (:yes :target _mc-get-input)
   (:otherwise :target _mc-process)))
```

Here, we first reset the conversation, meaning we clean the conversation object and erasing the window pane, then we print one of the 4 sentences, selecting one at random, then if the answer starts with "rien" we sleep, if it is negative, we sleep, if it is affirmative we go get a new input, otherwise we consider that the input was something to be processed directly. In this example the options :yes and :no will work for French, English, or German. For other languages, one can use :starts-with or :patterns to locate negative or positive answers.

Another example from the EXPLAIN conversation is the following:

```
(defstate
  _exp-entry-state
  (:label "Explain dialog entry")
  (:entry-state _explain-conversation)
  (:explanation "The data is cleaned prior to look for an entry point.")
  (:transitions
   ;;remove useless words explain, define, or else...
   (:patterns ((* "définir" ?y)
               (* "définition" ?y)
               (* "expliquer" ?y)
               (* "expliquez" ?y)
               (* "que" "veut" "dire" ?y)
```

```
              ("c" "est" "quoi" ?y "?")
              (?y "c" "est" "quoi" *)
              )
             :keep ?y ; put a list of the binding value into context HAS-DATA
             :sub-dialog _print-concept-documentation-conversation)
   ;; we do not know how the user got here and keep everything
   (:otherwise :sub-dialog _print-concept-documentation-conversation)
   )
  )
```

Here the text from the user is checked for the presence of some words, matching part of the input. The ?y variable will contain the end of the sentence that will be transferred to the :input part of the FACTS area of the conversation object, before we transfer control to the PRINT CONCEPT DOCUMENTATION sub-dialog.

The following piece of code corresponds to a sorry state, when the PA could not find anything. It tells the master and quits (returns a failure):

```
(defstate
  _ghad-sorry
  (:label "Address failure.")
  (:explanation "We don't have the requested address.")
  (:text
   "Désolé, si vous recherchiez une adresse personnelle, je ne la trouve pas.~%")
  (:transitions (:failure))
  )
```

6.8.2 Using Staff Agents

The following example from the GET BIBLIO conversation uses the BIBLIO agent:

```
(defstate
  _get-biblio-entry-state
  (:entry-state _get-biblio-conversation)
  (:label "Get biblio dialog entry")
  (:explanation "Assistant is sending a free-style message to the BIBLIO agent.")
  (:send-message :to :BIBLIO :self PA_ALBERT :action :get-publications
                 :args `(((:data . ,(read-fact moss::conversation :input))
                          (:language . :fr))))
  (:transitions
   (:on-failure :target _gbd-dont-understand)
   (:otherwise
    :exec (omas::assistant-display-text
           PA_ALBERT (moss::make-print-list (read-fact moss::conversation :answer)))
    :success))
  )
```

The PA sends a message to the :BIBLIO agent using the content of the :input from the FACTS area of the conversation object, and waits for an answer. If the answer is a failure then a transition to the Don't understand state is done, otherwise the content of the :answer area of FACTS is printed and a success is returned.

The following example does the same thing but the message contains a pattern specifying how the answer should be structured. The concepts and properties are taken from the PA ontology.

```
(defstate
  _get-biblio-entry-state
  (:entry-state _get-biblio-conversation)
  (:label "Get biblio dialog entry")
  (:explanation "Assistant is sending a free-style message to the BIBLIO agent.")
  (:send-message :to :BIBLIO :self PA_ALBERT :action :get-publications
                 :args `(((:data . ,(read-fact moss::conversation :input))
                          (:language . :fr)
                          (:pattern . ("personne"
                                       ("nom")
                                       ("prénom")
                                       ("adresse"
                                        ("domicile"
                                         ("rue") ("code postal") ("ville")))))))))
  (:transitions
   (:on-failure :target _gbd-dont-understand)
   (:otherwise
    :exec (omas::assistant-display-text
           PA_ALBERT (moss::make-print-list (read-fact moss::conversation :answer)))
    :success))
  )
```

6.8.3 Executing Some Piece of Code

If one has to do some light computation, then it is possible to execute some sequence of code either before the question is asked to the master, or prior to a transition.

The first example shows how to set up flags so that the reader does not use periods or question marks to terminate the input:

```
(defstate
  _pa-get-answer
  (:label "obtain master's answer to an ask message.")
  (:explanation "Record text typed into the master pane until the DONE button is ~
                 pushed. We must disable . and ? terminations.")
  ;; set global so that we can have . and ? in the text
  (:execute-preconditions
   (let ((win (car (has-input-window moss::conversation))))
     (when win
       (setf (omas::pass-every-char win) t)))
   )
  (:question-no-erase "~% ...Tapez votre réponse puis cliquez sur le bouton Terminer.")
  (:answer-analysis)
  )
```

This is done by the :execute-preconditions option.

The second example shows how code can be executed prior to a transition:

```
(defstate
  _pa-destroy
  (:label "Destroy ASK message.")
  (:explanation "Master does not want to answer the message nor keep it. We ~
```

```
                remove it from the list and return to the task with an :abort~
                mark, so that it will exit without answering.")
  (:transitions
   (:always
    :exec
    (let ((agent (car (has-agent moss::conversation))))
      ;; discard the message from the list, assuming it is still selected
      (omas::ASSISTANT-DISCARD-SELECTED-TO-DO-TASK agent)
      ;; clean up conversation
      (replace-fact moss::conversation :todo-message nil))
    :success))
  )
```

6.8.4 Complex Answer Analysis

When an answer from the master or from other agents is complex to analyze, then one can manually write an =answer-analysis method that will do the processing. This feature has been demonstrated in the Main Conversation dialog.

6.9 Some Problems

6.9.1 Input Text Segmentation

One of the problems that occurs in particular in the Chinese context is the problem of input segmentation. The input data is normally segmented using the space and punctuation marks, which cannot be done so easily in the Chinese contexts since words are not separated. Thus, a segmentation module should be inserted into the get-input function from the master. This cannot be done at the application level and requires an internal modification.

6.9.2 Overall State of the ALBERT-DIALOG.lisp File

The file has been developed in several steps and the code contained in this file is not always very clean. It could use a serious uplifting.

6.10 Syntax of the defstate Macro

The defstate macro is used to simplify the programming of dialogs. Its role is to create a state object and to synthesize the =execute and =resume methods attached to such a state. Normally, when the *crawler* function traverses the dialog graph and arrives at a specific state, it will tell something to the master or ask something (=execute method), wait for the answer, and then analyze the answer (=resume method). The defstate macro allows specifying such a behavior easily in most cases.

There are however some cases where the behavior at a given case may be different, e.g. if we analyze a previously obtained input directly, or if we want to print something and not wait, or if we want to ask another agent (staff agent of other), or if we want to do a very specific analysis of the master's input. In some cases the defmacro cannot synthesize the required code and an escape mechanism consists in writing a specific =answer-analysis method manually, or, if things are really complex to write the =execute and =resume methods manually.

This paragraph gives the current syntax of the defstate macro. The options may change in time since the conversation language they define is not really stable yet. Some options of the macro apply

to the =execute method, others apply to the =resume method. However, options may be given in any order, thus one must be aware of the time at which options will apply.

Finally, the code produced at each state applies mainly to the content of the conversation object, but may refer to any part of the OMAS system. The conversation object is a MOSS object, the structure of which is described in Table 6.1. Note that all properties start with the prefix MOSS-. This is to avoid potential conflicts with user-defined concepts or properties.

Table 6.1: Structure of the conversation object

Property	att/rel	role
MOSS-OUTPUT-WINDOW	att	interface window
MOSS-INPUT-WINDOW	att	interface window
MOSS-INITIAL-STATE	rel	pointer to the entry state
MOSS-STATE	rel	current-state of the conversation
MOSS-AGENT	att	PA ID implied in the conversation
MOSS-DIALOG-HEADER	rel	main conversation header
MOSS-SUB-DIALOG-HEADER	rel	current sub-conversation
MOSS-FRAME-LIST	att	list of return addresses of sub-dialogs
MOSS-TEXT-LOG	rel	record conversation texts
MOSS-TASK-LIST	rel	list of potential tasks for dialog
MOSS-TASK	rel	current task
MOSS-GOAL	rel	current goal (if any)
MOSS-FACTS	att	FACT base structured as an alternated list

Among all the properties of conversation the FACTS attribute is specially important since it is used to store and retrieve all kinds of informations during the dialog. Two functions are used to so that:

- replace-fact to insert a new value
- read-fact to retrieve an old value

6.10.1 Global Syntax

The global syntax is:

```
(defstate <state-variable> <options>*)
```

where

- state-variable points to the state object (by convention it will start with an underscore (e.g. _new-state) and should be declared as a global variable (see the examples in the ODIN-MAIL/ALBERT-DIALOG.lisp file) or be part of the :states option of the def-subdialog macro;
- options are expressed as a list starting with a keyword, e.g. (:execute (print moss::conversation)). Two options are useful, :label that should be short and label the state, and :explanation that gives a short description of what happens at this state.

Example:

```
(defstate
  _pt-brush
```

```
(:label "bookkeeping")
(:explanation "remove item from FACTS.")
(:execute (replace-fact moss::conversation :message nil))
(:transitions
 (:always :success))
)
```

The Backus-Nauer form of the defstate syntax is given in Table 6.2.

6.10.2 Global Options

They are:

- :label associated with a short text (string) that gives a title to the state;
- :explanation associated with a text (string) that gives an explanation of what happens in this particular state.

6.10.3 Options for the =execute method

The execute method should have a minimum of executable code. It is normally used to print a text or ask a question. The options are:

- :clear-all-facts clears the content of the FACTS base (removes everything)/
- :clear-fact item sets the value associated with item to NIL. Usually item is a keyword.
- :clear-facts item-list same as clear-fact but takes a list of items.
- :answer-type, can be used to specify the type (performative) of the answer message that will be returned.
- :eliza, call to ELIZA. We expect an answer from the master.
- :execute-preconditions, escape mechanism to execute Lisp code directly in the =execute method, e.g. to set flags.
- :question {:no-erase}, the associated value is either a string or a list of strings. When we have a list of strings one is chosen randomly for printing. Once the question is asked, the user is supposed to answer. The answer is inserted into the INPUT area of the FACTS slot if the conversation object that is the link between the external system and MOSS. When :no-erase is specified, the output screen is not erase before printing the question.
- :reset-conversation, resets internal conversation variables when restarting a dialog.
- :send-message args, sends a message to other agents. Args are those for an OMAS message. An answer is expected. A timeout may be specified.
- :send-message-no-wait args, sends a message to other PAs. Since the message is similar to an e-mail, we do not wait for the answer in the dialog (not in the process).
- :text {:no-erase}, prints text e.g. prior to asking a question.
- :text-exec expr, if expr is a list starting with format, executes it before printing it. Uses the =display-text method.
- :wait, sets OMAS to wait for master's input. Presumably the question has been asked in a previous state.

Table 6.2: Grammar of the defstate macro

Clause		Content
<state definition>	::=	(defstate <state name><global option><state options> ∗)
<state name>	::=	_<symbol>
<global option>	::=	(:label <string>) { (:explanation <string>)}
<state options>	::=	<execute option> ∗ <resume option>
<execute option>	::=	(:answer-type <performative>) |
		(:clear-all-facts)
		(:clear-fact <item>)
		(:clear-facts (<item>)+)
		(:eliza) |
		(:execute-preconditions <Lisp code>) |
		(:question <string>*) |
		(:question-no-erase <string>*) |
		(:reset-conversation)
		(:send-message <message args>) |
		(:send-message-no-wait <message args>) |
		(:text :no-erase <string>) |
		(:wait)
<resume option>	::=	(:answer-analysis) |
		(:execute <Lisp code>) |
		(:transitions <transition clause>*{<otherwise clause>})
<transition clause>	::=	(:always <action clause> ∗ <transition>) |
		(:contains (<string> ∗ <action clause> ∗ <transition>) |
		(:empty <action clause> ∗ <transition>) |
		(:no <action clause> ∗ <transition>) |
		(:on-failure <action clause> ∗ <transition>) |
		(:patterns <pattern list><action clause> ∗ <transition>) |
		(:start-with (<string> ∗ <action clause> ∗ <transition>) |
		(:text <expr><action clause> ∗ <transition>) |
		(:yes <action clause> ∗ <transition>) |
<otherwise clause>	::=	(:otherwise <action clause> ∗ <transition>) |
<action clause>	::=	:display-answer |
		:format-answer <format-function> |
		:keep <pattern variable> |
		:print-answer <print-function> |
		:replace <value> |
		:set-performative <performative> |
<transition>	::=	:failure |
		:reset |
		:subdialog <dialog header> {:failure <state name>} {:sucess <state name>}|
		:success |
		:target <state name>

6.10.4 Options for the =resume Method

The options can be organized in three groups : conditional options, intermediate options (actions), and transition options.

Conditional Options

- :always, means that the clause will always be executed.
- :contains list-of-words, efficient way of checking whether the input contains one of the words of the list.
- :empty, applies if after having processed the input, there is nothing left.
- :no, applies if the answer is a negative word. Does not work for Asian languages or languages where negation consists of restating the original question negatively.
- :on-failure, tests if the returned value from the master or from the agents was a failure by checking if the content of the FACT/INPUT area.
- :otherwise, always fire (should be the last option).
- :patterns <pattern>* <action>*, tests if the input contains patterns ELIZA style. If not, discards the clause. If so, executes the rest of the clause. This is an expensive option.
- :rules <pattern>* <answer>* <action>* tries to apply a set of rules. If one applies, then computes an answer ELIZA style, should use :display-result to display
- :starts-with list-of-words, efficient way of checking whether the input starts with one of the words in the list-of-words.
- :test expr, evaluates the Lisp expression to determine if the clause applies.
- :yes, applies if the answer is a positive word. Does not work for Asian languages or languages where agreement consists of restating the original question.

Intermediate Options or Actions

- :display-answer, prints the content of the FACTS/ANSWER slot of the conversation object.
- :display-result, displays the sentence resulting from applying the set of rules to the input; uses =display-text directly.
- :exec expr executes the expr in the context of =resume.
- :format-answer format-function, formats the content of the FACTS/ANSWER area applying the format-function to produce a string, uses then the =display-text method. It is the responsibility of the format-function to produce HTML strings when the answer must be shipped to the web server. format-answer takes 2 arguments: the first one is the answer the second one if t indicates that the result needs to be an HTML string.
- :keep variable, applies in the context of a the :patterns option and replaces the content of the FACTS/INPUT slot with the value associated with the pattern variable.
- :print-answer print-function, applies the function to the content of the FACTS/ANSWER slot. Deprecated: better to use format-answer.

- :replace list-of-words, replaces the current content of the FACTS/INPUT slot by the list-of-words.
- :set-performative performative, sets the content of the FACTS/PERFORMATIVE slot to the specified performative :request, :command, :assert

Transition Options

- :failure, returns from the current sub-dialog with a :failure tag.
- :reset, restarts the current sub-dialog at the entry state.
- :subdialog dialog-header {:failure state} {:success state}, starts a subdialog that should return either :failure or :success. The :failure and :success sub-options specify the target state. If none are mentioned, the returned value is transmitted to one more return level.
- :success, returns from the current sub-dialog with a :success tag.
- :target state, specifies the transition state.

6.11 Note Concerning Sending Messages

Care must be exercised when the programmer wants to send a request message to another agent from within the dialog. During the dialog the PA is not executing any specific task. If we send a message directly, when the answer comes back OMAS will not know where to put it, since a request is associated with a task. Thus, sending request is done by sending an internal message to the PA with :request as an :action and the content of the message to send as the :args data. OMAS will then create a task for the PA, send the external request message, recover the result and post it in the FACTS/ANSWER area of the conversation object. Another solution is to create a task manually, but this is not advised.

Note that this is done automatically by the :send-message option. Care must be exercised when writing an =answer-analysis method.

6.12 Simple Dialogs - defsimple-subdialog Macro

Some dialogs are very simple. One in particular is fairly prototypical. It consists in reading the input, sending it to a specific agent and printing the answer(s). Such a dialog requires two states: an entry state and a sorry state in case the task could not be executed.

For example consider the following dialog.: we want to obtain some information about financing research programs with another country. We only need to send a :get-financing message to the :FINANCING agent, which can be written:

```
(moss::defsimple-subdialog "get-financing" "_gfi"
     :explanation "Master is trying to obtain info about a financing program."
     :from PA_HDSRI :to :FINANCING :action :get-financing
     :language :fr
     :pattern ("financement" ("pays")("titre")("date limite") ("URL"))
     :sorry "- Désolé, je ne trouve pas le programme de financement demandé."
     :print-fcn #'print-financing
     )
```

"get-financing" is the name of the conversation and will create the _get-financing-cnversation; "_gfi" is the prefix for specifying the state variables. The PA will send a message to the :FINANCING agent with action :get-financing and arguments the list of words found in FACTS/INPUT area of

the conversation object. If the answer is a success, it will print the result otherwise it will print the sorry message. We need to provide a printing function, print-financing, taking advantage of the answer patterns specified in the :pattern option. Specifying a pattern is not compulsory. If no pattern is specified, the answer will be a list of strings prepared by the FINANCING agent, each string dealing with a program.

6.13 PA Communications

The dialog system needs to be connected to the application. To do so one needs to specify where do the data come from (input channel) and where the output must be printed (output channel). In the OMAS environment each PA has a conversation slot pointing to the conversation object.

Vocal input requires a system for transforming the utterances into strings. The Dragon system was used in the ACL version of OMAS. The connection between the voice module and OMAS is done through a socket.

6.14 Creating a Foreign Personal Assistant

The easiest way to create a foreign assistant is to modify an existing one, say ALBERT. Some changes are required to various parameters and files.

- First, the target language must be part of the list of languages supported by MOSS. The allowe languages are currently defined by a global parameter:

  ```
  (defParameter moss::*language-tags* '(:cn :en :es :fr :it :jp :lu :pl :unknown)
    "allowed languages")
  ```

  However, if one wants to use Chinese or Japanese language, one must provide a segmentation routine.

- The PA-TASKS.lisp file must be rewritten using the new language.
- The PA-ONTOLOGY.lisp file should be translated and upgraded using the new language.
- The PA-DIALOG.lisp file must be translated and upgraded.

Note on Asian languages: One of the problems that occurs in particular in the Chinese context is the problem of input segmentation. The input data is normally segmented using the space and punctuation marks, which cannot be done so easily in the Chinese or Japanese contexts since words are not separated. Thus, a segmentation module should be inserted into the get-input function from the master. This cannot be done at the application level and requires an internal modification of OMAS.

6.15 The Default Detailed PA Interface

A more detailed PA interaction window can be called when messages are directly addressed to the user and the user wants to see what the PA did with such messages. The "big window" button allows to switch from the simple window to the detailed window shown 6.7.

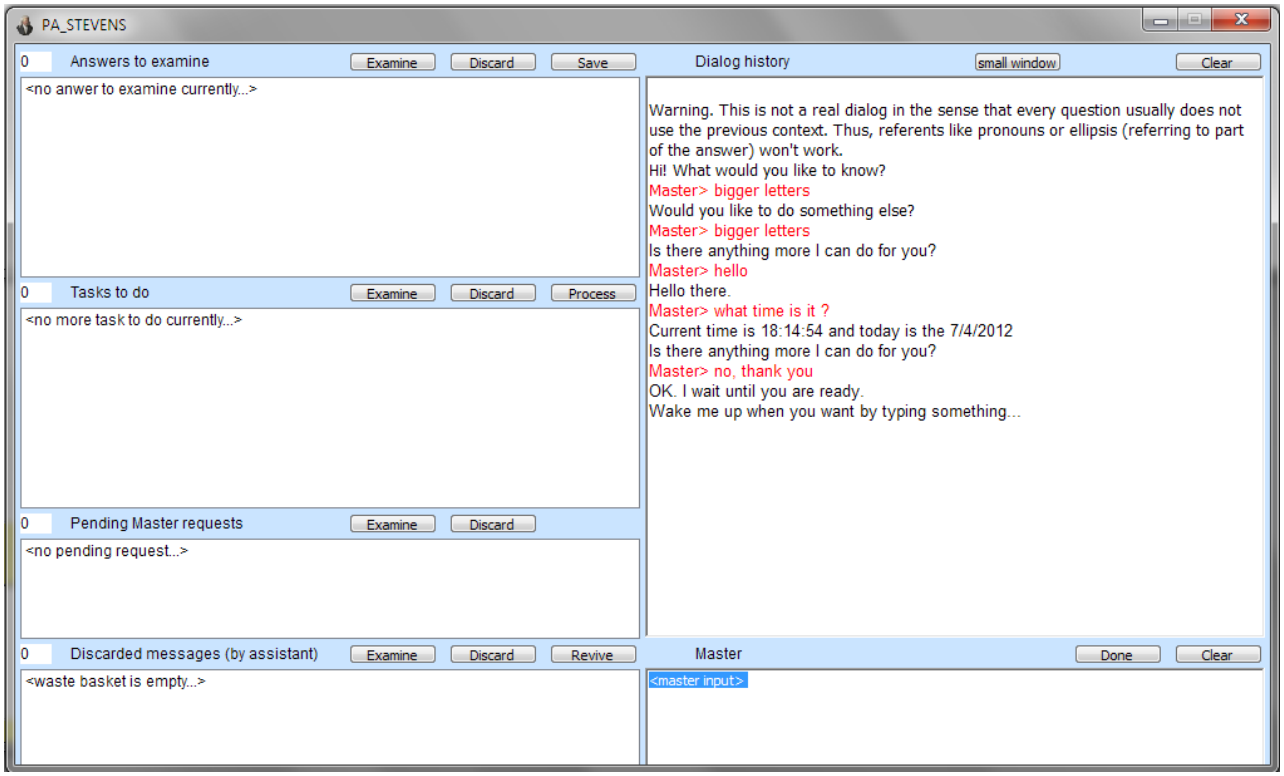

Figure 6.7: OMAS detailed interaction window

6.16 PA Default Interaction Window

6.16.1 Mechanism

Communications with the PA

The right part of the interface has two panes:

- **Dialog history**: an area in which the PA prints questions or answers.
- **Master**: an area in which the master (user) inputs data or requests to the PA.

External Communications

The left part of the interface includes four areas:

- **Answers/Info**: an area containing answers to requests done by the PA on behalf of the master, and the history of the dialog.
- **Tasks to do**: the list of requests that are asked to the master and that the agent could not process by itself.
- **Pending requests**: requests that have been asked by the master and that did not receive an answer yet.
- **Discarded messages**: the waste basket into which the assistant has thrown irrelevant messages.

The following sections detail the use of the different areas.

6.16.2 Answers/Info

The Answers/Info area displays messages that are either answers to questions that the master asked previously or information messages corresponding to the PA skill `:tell`.

Message Summary

Each line corresponds to a single message, like in an email browser. The displayed information indicates the day and hour the message has been received, the urgency, the sender identity, and the object of the message.

Example

```
08/09/12/ 9:26:40/ NORMAL/ Meteo
```

Possible Actions

The answers/info area has three buttons:

- Examine: to display the content of a message into the Assistant area;
- Discard: to remove a message from the list;
- Save: to save the content of the message somewhere in the memory.

Viewing Dialog

In the French version of a PA examining the dialog prints the following information into the assistant pane:

```
Expéditeur : <USER>
Urgence : Normal
Objet : Méteo

 Il va pleuvoir

Voulez-vous garder ce message ?
(Do you want to save this message?)
```

If the answer is "oui" (yes) the message is kept, otherwise the message is discarded.

6.16.3 Tasks to Do

The tasks to do area displays messages that have been received by the PA and correspond to questions to the master that need to be answered. The corresponding PA skill is `:ask`.

Message Summary

Each line corresponds to a single message, like in an email browser. The displayed information indicates the day and hour the message has been received, the urgency, the sender identity, and the object of the message.

Example

```
08/09/12/ 9:26:23/ URGENT/ Your mother's birthday
```

Possible Actions

The tasks to do area has three buttons:

- Examine: to display the content of a message into the Assistant area;
- Discard: to remove a message from the list;
- Process: to display the content of the message and start a dialog to process it.

Processing Dialog

In the French version of a PA the processing dialog goes as follows:

```
Voulez-vous répondre à ce message ?
(Do you want to answer this message?)
```

If the answer is "non" (no) then

```
Voulez-vous garder ce message pour le traiter plus tard ?
(do you want to keep this message for later processing?)
```

If the answer is "non" (no), then the message is deleted. Otherwise the message is kept in the list.

If the answer to the first question is "oui" (yes), then

```
Tapez votre réponse puis cliquez sur le bouton DONE.
(Type in your answer and click the DONE button.)
```

The system allows you to input a text message as long as you want until you click the DONE button. After you clicked the DONE button the answer is returned to the sender of the message.

6.16.4 Pending Requests

The pending requests area displays messages corresponding to requests that have been sent by the master but have not yet been answered. I.e. there is a process waiting for the answer.

Message Summary

Each line corresponds to a single message, like in an email browser. The displayed information indicates the day and hour the message has been received, the urgency, the sender identity, and the object of the message.

Example

```
08/09/12/ 9:26:40/ NORMAL/ Air France ticket to RIO
```

Possible Actions

The tasks to do area has three buttons:

- Examine: to display the content of a message into the Assistant area;
- Discard: to remove a message from the list;

Removing the message from the list kills the corresponding task and sends an abort message in broadcast mode.

Viewing Dialog

In the French version of a PA the processing dialog goes as follows:

```
Destinataire : Agence de voyage
Urgence : Normal
Objet : Air France Réservation pour RIO

Bonjour,
Voudriez-vous me dire si la réservation pour le vol sur RIO a été faite ?
Merci.

Voulez-vous garder ce message ?
(Do you want to save this message?)
```

If the answer is "oui" (yes) the message is kept, otherwise the message is discarded.

6.16.5 Discarded Messages

The Answers/Info area displays messages that have been discarded by the PA. The master has the possibility to retrieve them from the waste basket.

Message Summary

Each line corresponds to a single message, like in an email browser. The displayed information indicates the day and hour the message has been received, the urgency, the sender identity, and the object of the message.

Example

```
08/09/12/ 11:21:40/ NORMAL/ Air France ticket to RIO
```

Possible Actions

The tasks to do area has three buttons:

- Examine: to display the content of a message into the Assistant area;
- Discard: to remove a message from the list;
- Revive: to remove a message from the the waste basket and insert it into the Answers/Info area or the Tasks to Do area;

Viewing Dialog

In the French version of a PA the processing dialog goes as follows:

```
Expéditeur : PayPal
Urgence : Normal
Objet : Your account

Info about your account has been lost.
Please send you password again.

Voulez-vous récupérer ce message ?
(Do you want to recover this message?)
```

If the answer is "oui" (yes) the message is extracted from the waste basket, otherwise the message is left there.

6.17 User-Defined Interaction Window (Windows)

In some cases it is convenient to specify a particular interaction window to replace the stansard default interface window. The mechanism has been implemented so far in the ACL environment. This section gives an example of user-defined interaction window.

6.17.1 Specifying an Alternate Interaction Window

Specifying a particular interaction window is done by declaring it in the defassistant macro:

```
(omas::defassistant :PATIENT :language :EN :interface make-patient-window)
```

The interface option names a function that will be called by the converse process for creating the window. It is a function of no argument to be defined in the agent file.

6.17.2 The Window Object

For bookkeeping purposes one must declare the class of the new interface window as a subclass of the omas::omas-assitant-panel class, e.g.

```
(defclass patient-window omas::omas-assitant-panel) ())
```

This ensures that the internal OMAS structures are upgraded when creating or destroying the window. The class inherits 3 specific slots: agent, answer-to-task-to-do, and pass-every-char.

6.17.3 Necessary Methods

In order to hook the new window to the dialog handling system, one must define several specific methods that are called by OMAS when the PA wants to print or add something into the interface or read from the interface

- a MOSS method located in the MOSS package (moss::activate-input) to activate the window, sow it and select the input area
- a MOSS method located in the MOSS package (moss::display-text) to display information

The rest depends on the application.

6.17.4 Connection with the Dialog Mechanism

Provided the necessary methods are defined, the connection with the dialog mechanism is automatically activated. Thus, the dialog starts as soon as the agent has been loaded.

The input callback must then call the omas::assistant-process-master-text function to process the input text.

6.17.5 Example

The following example is extracted from a project done with Prof. Hattori from Ritsumeikan University Kyoto.

PATIENT is a PA in charge of discussing with its master through a specific interface shown Fig. 6.8. Questions and information are displayed in the interface and a menu allows selecting conversation topics.

Figure 6.8: Caap specific Personal Assistant interaction window

Defining the Agent

We define an English speaking PA with a special interface.

```
(omas::defassistant :PATIENT :language :EN :interface make-patient-window)
```

Defining the Interface Window

The window inherits from the OMAS PA pane.

```
(defClass patient-window (omas::omas-assistant-panel)
  ((agent :accessor agent :initform nil))
  (:documentation "special patient interaction window")
  )
```

I/O methods

Activating the input area(ACL Windows environment):

```
;;;---------------------------------------- (PATIENT-WINDOW) MOSS::ACTIVATE-INPUT

(defMethod moss::activate-input ((win patient-window) &key erase)
  "prints a text into the output pane of the window.
Arguments:
   erase (key): if true erase the content, otherwise select it
Return: unimportant."
  ;; activate the user panel
  ;; the user will enter its data that will be transferred into the to-do
  ;; slot, waking up the resume process
  (cg:select-window win)
```

```
  ;; erase pane
  (moss::display-text win "" :erase erase)
  ;; activate the master pane
  (cg:set-focus-component (cg:find-component :input-pane win))
  )
```

For displaying text the method uses some of the options of the standard display method.

```
(defMethod moss::display-text ((win patient-window) text &key
                               (erase t) newline
                               final-new-line (header "") clean-pane
                               (color moss::*black-color*)
                               &allow-other-keys)
  "prints a text into the output pane of the assistant panel.
Arguments:
   text: text to display (must be a simple string)
   clean-pane (key): erase the assistant pane in all cases
   erase (key): if t, erase previous text, otherwise append to it
   final-new-line (key): if t, add newline command at the end of text
   header (key): a string that will be printed in front of the text
                 (default \"\")
   newline (key): it t, add a new line in front of the header
Return:
   nil."
  (let* ((output-pane (omas::%find-named-item :answer-pane win))
         )
    (unless (and (stringp text)(stringp header))
      (error "text ~S and header ~S should be strings." text header))

    (if final-new-line (setq text (concatenate 'string text "~%")))
    (if newline (setq header (concatenate 'string "~%" header)))

    ;; because text or header could contain format directives, we process it
    (setq text (format nil (concatenate 'string header text)))

    (when clean-pane
      (omas::%set-value output-pane "")
      (return-from moss::display-text))

    (format t "~%; patient moss::display-text /text: ~S" text)
    (if erase
      (omas::%set-value output-pane (format nil text))
      ;; add more text
      (omas::%set-value output-pane
                (format nil "~A~A" (omas::%get-value output-pane) text))
      )
    nil))
```

In this code, macros starting with a % sign can be used in ACL and MCL environment.

Creating the Interface Window

This is fairly standard ACL code.

```
(defUn make-patient-window (SA &key (left 200)(top 200)
                                    (width *table-width*)(height *table-height*)
                                    (topic-list *topic-list*)
                                    (name :patient-window)
                                    (class 'patient-window)
                                    &allow-other-keys)
  (declare (ignore lrest SA))
  (format t "~%; make-patient-window / process ~S" mp:*current-process*)
  (let* ((frame (cg:make-window name
                                :class class
                                :owner (cg:screen cg:*system*)
                                :title "TOPICS FOR DISCUSSION"
                                :background-color cg::yellow
                                :exterior (cg:make-box-relative left top
                                width (+ 18 height))
                                ;:dialog-items (make-patient-dialog-items)
                                :resizable nil
                                :maximize-button nil
                                :scrollbars nil
                                :state :shrunk))
         topic-label box)
    ;; try to add topic area
    (dotimes (kk *max-nb-of-topics*)
      (setq box (compute-box kk))
      (setq topic-label
            (make-instance 'cg:scrolling-static-text
              :name (intern (format nil "TOPIC-~S" kk) :keyword)
              :value (nth kk topic-list)
              :background-color (cg:make-rgb :red 200 :green 200 :blue 255)
              :left (cg:left box) :top (cg:top box)
              :width (cg:width box) :height (cg:height box)
              :on-click 'select-topic-on-click
              :on-mouse-in 'change-color
              :on-mouse-out 'restore-color
              ))
      (cg:add-component topic-label frame))
      ;;; etc.
      ...)
```

Inputing Data

The callback corresponding to the input area is as follows:

```
;;;------------------------------------------------------------- INPUT-ON-CHANGE

(defUn input-on-change (item text old-value)
  "Called whenever the content of the input pane (a string) changes.
   We check whether the new char is a period or a question mark
```

```
 if so we terminate input and process it."
(declare (ignore old-value))
;;extract last char and text before the last char, when text is not empty
(let ((char (if (> (length text) 0) (char text (1- (length text)))))
      (text (if (>= (1- (length text)) 0)
              (subseq text 0 (1- (length text)))
              "")))
  ;; analyse situation:
  ;; - if char is nil do nothing
  ;; - if char is question mark selet the text and call process-master-text
  ;;   adding question mark to the text
  ;; - if char is a period, then select the text and call process-master-text
  ;; - if text is a linefeed, beep ??
  ;; - otherwise do nothing.
  (cond
   ;; a null char should not appear unless we erase area
   ((null char)) ; do nothing
   ;; if the char is a question mark, insert a space
   ((member char (getf omas::*question-markers* *language*))
    ;; select text
    (cg:set-selection item 0 (1+ (length text)))
    ;; add a " ?" at the end of the text
    (omas::assistant-process-master-text
     (agent (cg:parent item))
     (format nil "~A ~A" text char)))

   ;; when a period, end of the sentence (ACL does not know #\Enter!)
   ((member char (getf omas::*full-stop-markers* *language*))
    ;; select text
    (omas::%select-all item)
    (omas::assistant-process-master-text (agent (cg:parent item)) text)
    )
   ((char-equal '#\Linefeed char)
    (cg:beep))
   ))
;; return t
t)
```

One must notice the call to omas::assistant-process-master-text that will process the text input.

6.18 Web Interface (Windows)

The possibility of accessing the data from the web has been added in May 2011 (OMAS-MOSS v8.1.3). It is an experimental simple mechanism using the Allegro aserve web page server.

6.18.1 The Web Server

Using the web server requires loading the omas-web file in the agent definition file:

```
(load-omas-file "omas-web")
```

The server is activated from the OMAS.WEB package, created as recommended in the ACL aserve documentation.

Starting the server is done by calling omas::start-web-server with a port number (default is 8000):

```
(omas::start-web-server 50002)
```

Publishing pages is done with the ACL publish function called in the OMAS.WEB package. Various functions can be created calling publish to define web pages. We use HTML forms to input data through web pages.

6.18.2 Getting Form Data

The function get-data (in the omas.web package) can be called to recover data from the different forms, send a message to a PA skill and wait for the processing of this message with a timeout.

6.18.3 Synchronizing the Agents

Since we are operating in a multi-processing environment, synchronization is done by means of a gate. Once the data is received from the processing of the different agents, the answer is put into a global variable and the gate is opened.

6.18.4 Example

The following example is taken from the HDSRI application. The HDSRI PA handles requests from several users concerning requests about international relationships. Requests can come from an e-mail but to the user this may be slow since the answer time is proportional to the polling time of the mailbox. A faster approach consists in setting up a web server. To do so we use Allegroserve.

We define first several skills to give to HDSRI, although we could do without by systematically launching the web server when the agent is loaded.

The PA file must load the omas web file that is not loaded by default. This is done by inserting the following command at the beginning of the PA file:

```
(load-omas-file "omas-web")
```

The skills are then defined as:

```
;;;============================================================================ skill
;;;                                  :START-WEB
;;;============================================================================

(defskill :START-WEB :HDSRI
  :static-fcn static-start-web)

(defUn static-start-web (agent message)
  (declare (ignore message))
  (omas::start-web-server 8000)
  (static-exit agent :done))

;;;============================================================================ skill
;;;                                   :STOP-WEB
;;;============================================================================

(defskill :STOP-WEB :HDSRI
```

```
  :static-fcn static-stop-web)

(defUn static-stop-web (agent message)
  (declare (ignore message))
  (omas::stop-web-server)
  (static-exit agent :done))
```

The two skills can be used to start and stop the web server. To populate the server one needs to publish a page, which can be done with the following skill:

```
;;;============================================================================ skill
;;;                              :PUBLISH-WEB-PAGE
;;;============================================================================

(defskill :PUBLISH-WEB-PAGE :HDSRI
  :static-fcn static-publish-web-page)

(defUn static-publish-web-page (agent message)
  (declare (ignore message))
  (publish-web-page)
  (static-exit agent :done))
```

Now we need to define the publish-web-page function:

```
(defUn hdsri::publish-web-page (&aux answer)
  (publish
   :path "/queryform"
   :content-type "text/html"
   :function
   #'(lambda (req ent)
       (let ((user-input (cdr (assoc "user-input" (request-query req) :test #'equal))))
         (with-http-response (req ent)
           (with-http-body (req ent)
             (if* user-input
                  ;; here we got some question
                  then ; form was filled out, we must process the question (call a function to
                  (setq answer (get-answer user-input :HDSRI))
                  ;; print the page with the result underneath
                  (answer-form user-input answer)
                  ;; we come here the first time around since
                  else ; put up the form
                  (ask-query-form)))))))))
```

The above function is inspired by the example from ACL. It calls two functions:

- ask-query-form, called the first time the page is requested;
- answer-form, to post the result and wait for a new request.

The two auxiliary functions use the html macro from ACL and are detailed here:

```
(defMacro ask-query-form ()
  '(html
```

```
    (:html
     (:head (:title "HEUDIASYC : Relations Internationales"))
     (:body
      ((:form :action "queryform")
       (:h1 "HEUDIASYC : Relations Internationales")
       (:b "Question ? ")
       :br
       ((:textarea :name "user-input" :rows 3 :cols 100))
       :br
       ((:input :type "submit" :value "Envoyer"))
       )))))

(defMacro answer-form (user-input answer)
  `(html
    (:html
     (:head (:title "HEUDIASYC : Relations Internationales"))
     (:body
      (:h1 "HEUDIASYC : Relations Internationales")
      (:b "Réponse à : ")
      (:princ ,user-input)
      :br
      :br
      (:princ ,answer)
      ((:form :action "queryform")
       :br
       (:b "Question ? ")
       :br
       ((:textarea :name "user-input" :rows 5 :cols 100))
       :br
       ((:input :type "submit" :value "Envoyer"))
       )))))
```

Note the get-answer function, used to synchronize with the returning messages from the MAS. The function is internal to OMAS but is shown here to help understanding the synchronization process:

```
;;;------------------------------------------------------------------- GET-ANSWER

(defun get-answer (text to-agent &key (gate *web-gate*) (timeout 30)
                         (action :process-web-message))
  "takes the input and sends a message to the to-agent. Since we are not an agent ~
    the sender is nil. We then wait on a global gate. The :args field of the send ~
    message has the folllowing format:
    ((:data . text)(:gate . gate)(:answer . answer)) where answer is a global ~
    variable in which to put the answer.
Arguments:
   text: a string that will be sent as the data part of the protocol
   to-agent: a keyword specifying the agent that will receive the message
   gate (key): a variable designating a synchronizing gate (default *web-gate*)
   timeout (key): the timeout delay (default 30s)
   action (key): the name of the skill to be activated (default :process-web-message)
Return:
```

```
 an html string representing the answer or a default error message on timeout."
(declare (special *web-gate* *web-answer*))
(mp:close-gate gate)

(let ((message
       (make-instance 'omas::message
         :type :request :to to-agent :date (get-universal-time)
         :action action
         :args `(((:answer . *web-answer*)(:data . ,text) (:gate . ,gate)))))
      )
  ;; reset the answer
  (setq *web-answer* nil)
  ;; send the message
  (send-message message)
  ;; wait according to timeout
  (cond
   ((mp:process-wait-with-timeout "waiting web processing" timeout
                                  #'mp:gate-open-p gate)
    (mp:close-gate gate)
    ;; answer should be some sort of html string
    *web-answer*
    )
   ;; otherwise we did not get an answer in time
   (t
    (make-html-error-page)))
  ))
```

Note that since we are in a multiprocessing environment, the function uses a gate for synchronization.

The web interface can be used to define forms for inputting data.

6.19 Email Interface (Windows)

The mail interface was used prior to the development of the Web interface. Indeed, the Web interface gives faster answers than e-mail, which tends to make the e-mail interface obsolete.

6.19.1 Mechanism

The email interface uses the smtp feature from the ACL library. Using the e-mail interface bypasses the dialog mechanism, calling a PA skill directly. In practice, the PA has a goal to examine the content of the mailbox periodically and if the mailbox contains some messages, it triggers the adequate skill.

Thus, to use the e-mail interface, one must define a goal for the PA, write the skill to process the incoming messages, and insert a command to load the smtp library at the beginning of the PA file.

6.19.2 Example

The example is taken from the HDSRI application that handles international relationships.

First, load the smtp library function b inserting the following line at the beginning of the PA file.

```
(load-omas-file "omas-email")
```

The define a goal that will be inserted at the end of the PA file (recommended).

```
;;; ================================ goal section ============================

(defgoal :PROCESS-EMAIL :HDSRI
  :mode :rigid
  :type :cyclic
  :period 20
  :goal-enable-fcn enable-process-email
  :script process-email-script)

(defUn process-email-script (agent)
  "sends a message to read the mailbox, extract HDSRI emails, answer them"
  (declare (ignore agent))
  (list
   (make-instance 'omas::message :type :request :from :HDSRI :to :HDSRI
                  :action :process-email
                  :args nil
                  )))

;;; return T to enable the goal NIL to inhibit it

(defUn enable-process-email (agent script)
  "always true, i.e. goal always active"
  nil) ; nil prevents firing...
```

The goal fires every 20 seconds and calls the process-email skill defined as follows:

```
(defskill :process-email :HDSRI
  :static-fcn static-process-email
  :dynamic-fcn dynamic-process-email
  )
```

Note that the skill has 2 functions, a static one and a dynamic one. The static function looks into the maibox to see if there are some mails. If so it processes them discarding junk mail.

```
(defUn static-process-email (agent message)
  "looks into the mailbox and processes first relevant email  if any"
  (declare (ignore message))
  (let (input message-sent? task-list raw-input from-address)
    ;; set first mailbox access
    (omas::set-mailbox-parameters "kappa.utc.fr" "hdsri" "hdsri-password")
    ;; go get first relevant message
    (multiple-value-setq (input from-address)
      (omas::get-next-email agent :pattern "HDSRI"))
    ;; if none quit
    (unless input
      (return-from static-process-email (static-exit agent :done)))

    ;;=== process message
    ;; function returns non nil if a subtask was created (i.e. email was
    ;; understood and a message was sent to a service agent) together with a list
    ;; of tasks that could apply (in case of failure) and the text content of email
```

```
    (multiple-value-setq (message-sent? task-list raw-input)
      (omas::process-single-email agent input))
    ;; if the message was not understood, try to return something helpful
    (unless message-sent?
      (omas::send-html-message agent *help-text* from-address
                               :subject "De la part de HDSRI")
      )
    ;; when the email input was understood and a message was sent to a service
    ;; agent, then we record the list of tasks in case the answer from the
    ;; service agent is a failure
    (when message-sent?
      (env-set agent task-list :next-tasks)
      (env-set agent raw-input :text))
    (env-set agent from-address :message-from)
    ;; if subtask was created dynamic skill will take care of it
    (static-exit agent :done)))
```

The static part of the skill first opens the mailbox by calling the omas::set-mailbox-parameters function:

```
(omas::set-mailbox-parameters "kappa.utc.fr" "hdsri" "hdsri-password")
```

This indicates that the server is kappa.utc.fr and gives the login and password.

It then calls the omas::get-next-email function that returns 2 values a message and a sender's IP when there is a relevant message.

```
(omas::get-next-email agent :pattern "HDSRI")
```

The pattern argument specifies that messages with HDSRI in the object line will be returned, all other messages will be discarded.

Then the omas::process-single-email function is called to process a singlemessage, i.e. to do the requested action (usually sending a message). If no message was sent, then an answer indicating that the message was not understood is returned to the user. Otherwise, the list of tasks, the unprocessed input text and the sender IP are recorded into the agent memory.

The dynamic part of the skill processes the returned result.

```
(defUn dynamic-process-email (agent message answer)
  "get results of task. If no result try next task. if no more task, quit."
  ;; when the message was processed, it may have determined several tasks.
  ;; If the result of the first task is a failure, then we must try the
  ;; next task. If there are no more tasks, then the message was not understood
  ;; and we should send an alert to the HDSRI manager and a message back to
  ;; the sender stating that the phrasing was not understood.
  ;; We then go process the next message in the mailbox
  (let ((next-tasks (env-get agent :next-tasks))
        message-sent? tasks-left message-from email-content)
    (cond
     ;; if task was a failure and there are tasks left
     ((and (moss::is-answer-failure? answer) next-tasks)
      ;; process next eligible task with same old message
      (multiple-value-setq (message-sent? tasks-left)
        (omas::process-next-task
```

```
       agent next-tasks
       `((:data . ,(env-get agent :text))(:language . :fr))))
    ;; if task was found and subtask was sent, get out, wait for subtask answer
    (when message-sent?
      (env-set agent tasks-left :next-tasks)
      (return-from dynamic-process-email))
    ;; otherwise, we had no eligible task left, go try for next message
    ;; clear first next tasks
    (env-set agent nil :next-tasks)
    )

   ;; here failure and no more tasks,
   ((moss::is-answer-failure? answer)
    ;; as a safety measure clear the next-tasks slot
    (env-set agent nil :next-tasks)
    ;; tell sender we did not understand the message
    (omas::send-html-message agent *help-text* (env-get agent :message-from)
                     :subject "De la part de HDSRI")
    ;; go check if there are emails left in the box
    )

   ;; here we have a bona fide answer
   (t
    ;; process answer, i.e. send it back to the caller, using MIME format
    (omas::send-html-message agent answer (env-get agent :message-from)
                     :subject "De la part de HDSRI")
    ;; clean the task list input and sender as a safety measure
    (env-set agent nil :next-tasks)
    (env-set agent nil :message-from)
    ;; and go check whether there are emails left in the box
    ))

  ;; Here, we check for other messages
  (loop
    ;; get-next-mail opens and closes the mailbox
    (multiple-value-setq (input message-from)
      (omas::get-next-email agent :pattern "HDSRI"))
    ;; if no more messages we are through and the mailbox has been emptied
    (unless input
      ;; clear task list as safety measure
      (env-set agent nil :next-tasks)
      (return-from dynamic-process-email (dynamic-exit agent :done))
      )
    ;; if something process it
    (when input
      ;; save return address
      (env-set agent message-from :message-from)
      ;; process email
      (multiple-value-setq (message-sent? tasks-left email-content)
        (omas::process-single-email agent input))
```

```
  ;; when process-single-mail returns nil, it means that no message was not
  ;; understood (no eligible task found)
  (when message-sent?
    ;; save content and tasks left
    (env-set agent email-content :input)
    (env-set agent tasks-left :next-tasks)
    ;; quit, answer will wake up dynamic skill again
    (return-from dynamic-process-email))
  ;; otherwise we could not find a task, send "not understood" message
  (omas::send-html-message agent *help-text* message-from
                           :subject "De la part de HDSRI")
  ;; clear task list
  (env-set agent nil :next-tasks)
  )
 ;; go see if there is another message left
 )))
```

Basically this skill implements the following strategy:

- if the task was a failure but there are other possible tasks to process the input, then the next possible task is called with the omas::process-next-task function;
- if the task was a failure and there are no more tasks, we send a failure message to the user;
- if thee was an answer, we return an HTML string to the user by calling the omas::send-html-message function:

```
(omas::send-html-message agent *help-text* (env-get agent :message-from)
                         :subject "De la part de HDSRI")
```

- then we enter a loop to find if there are any other message to process by keeping calling the omas::process-single-email function.

6.20 Voice Interface

There are several possibilities for giving a PA a vocal interface and of implementing the interface. We discuss the ones more in line with the OMAS approach. We assume that we have a speech-to-text and a text-to-speech software that runs in parallel with OMAS agents. There are essentially two ways of connecting a PA with the voice system: (i) directly using sockets; (ii) indirectly using messages.

6.20.1 Direct Socket Connection

The direct socket connection is justified whenever the user wants to be connected directly to her PA. The voice interface implements a direct communication between the voice component and the PA and is not meant to be heard by other PAs in the coterie. In that case a tight coupling makes sense. We thus use a direct UDP socket connection as shown Fig.6.9.

In Fig.6.9 the voice-input-port is the PA port receiving the string resulting from a speech to text conversion of the Voice System. The voice-output-port is the port of the voice system receiving the string to be converted to speech.

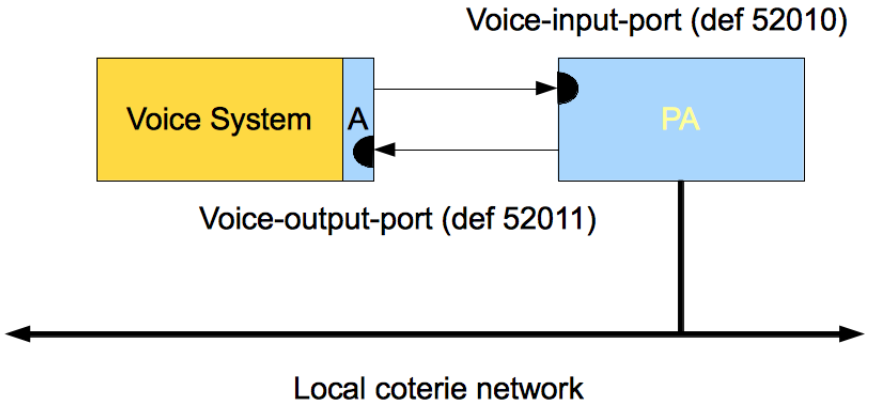

Figure 6.9: Direct connection using sockets

Implementation

To implement this solution, one must tell OMAS what are the input port (PA side) and the output port (voice component side) of the connection. This is done when creating the PA by using the :voice-input socket and :voice-output-socket options. If the options are not used then sockets 52010 and 52011 are used by default.

The code that deals with the messages on the OMAS approach is now described.

When the PA is created with a :voice t option, then the voice interface is initialized as follows:

```
;;;------------------------------------------------------------ NET-INITIALIZE-VOICE

(defun net-initialize-voice (agent)
  "This function opens a socket if necessary.
Arguments:
   agent: assistant agent controling the vocal interface
Return:
   :done"
  ;; create a receiving socket to be used on the PA side
  (unless (voice-input-socket agent)
    (setf (voice-input-socket agent)
      (socket:make-socket
       :type :datagram
       :local-port (voice-input-port agent)
       ))
    )
  ;; create a sending socket
  (unless (voice-output-socket agent)
    (setf (voice-output-socket agent)
      (socket:make-socket :type :datagram))
    )
  ;; set up receiving process unless it already exists
  (unless (voice-receiving-process agent)
    (setf (voice-receiving-process agent)
          (mp:process-run-function "Net Voice Receive" #'voice-receiving agent)))
```

```
  :done)
```

This function creates two sockets: (i) a receiving socket (voice-input-socket agent); and (ii) a sending socket (voice-sending-socket agent). The sockets are defined on the local machine and use a UDP protocol. The receiving socket is used by a function called voice-receiving that is essentially a loop:

```
;;;------------------------------------------------------------------ VOICE-RECEIVING

(defun voice-receiving (agent &aux text)
  "receives a message from the voice input and puts it into the to-do slot of the
agent."
  (loop
    (multiple-value-bind
          (raw-buffer size)
        (socket:receive-from (voice-input-socket agent)
                             (voice-max-message-length agent)
                             :extract t)
      ;; transform unicode string into internal string
      ;; truncate is added as a safety measure
      (setq text (excl::octets-to-string raw-buffer ;:external-format :fat
                                         :truncate t))
      ;; process the received string
      (omas::assistant-process-master-text agent text)
      )))
```

When the PA wants to send a string to the voice system it uses the net-voice-send function defined as follows:

```
;;;------------------------------------------------------------------ NET-VOICE-SEND

(defun net-voice-send (agent message)
  "sends a message to the voice interface using the UDP protocol.
Arguments:
   message: a String less than *max-message-length* bytes long
Return
   code returned by the send-to function."
  ;; check arg
  (unless (stringp message)
    (error "message should be a string rather than: ~S" message))
  (if (> (length message) (voice-max-message-length agent))
      (error "message too long:~%~S" message))
  ;; trace
  (format t "~%; net-voice-send / sending message to ~A:~S~%;  ~%;   ~S"
    (voice-ip agent) (voice-output-port agent) message)
  ;; send message
  (when (voice-output-socket agent)
    (socket:send-to (voice-output-socket agent) message (length message)
                    :remote-host (voice-ip agent)
                    :remote-port (voice-output-port agent))))
```

Setting the Stage

In order to activate the voice mechanism, one has to use the following options when defining the personal assistant:

```
(defassistant :albert :voice t :voice-input-port 4000 :voice-output-port 4001 ...)
```

Discussion

The direct connection approach is well suited for accessing a PA with minimal overhead. It inserts the string resulting from the speech-to-text conversion into the TO-DO slot of the PA, which wakes up the converse process managing the dialog. The string is then processed as if it had been typed into the master's pane of the interface window.

Then, because the voice connection is enabled, every time something is written to the assistant pane, it will be sent to the voice synthesizer.

Voice Component on a Different Machine

One can install the voice component on a different machine as long as it sends the messages to the right socket of the PA.

On the PA side it is necessary to specify the IP of the machine containing the voice component and its port number if different from the default one, e.g.

```
(defassistant :albert :voice t :voice-input-port 4000 :voice-output-port 4001
      :voice-ip "skopelos" ...)
```

This means that the PA will receive transcribed vocal messages on port 4000 and the voice component located on skopelos will receive strings to transform into speech on port 4001.

6.20.2 Message Connection

In this approach we get the string corresponding to the user's utterance and send it to the PA using an OMAS message. There are two possible ways of implementing it: (i) directly producing OMAS messages and sending them on the local coterie loop; or (ii) using a postman to produce the message.

Synthesizing OMAS Messages

The approach consists in wrapping the voice recognition software in a piece of software that will send an OMAS message on the local coterie loop. Then, the answer message will be analyzed and sent to the speech-to-text processor. The structure is described Fig.6.10.

In this approach the voice system is interfaced to the OMAS platform by a software layer that transforms the vocal input into an OMAS message. The message is then put onto the local coterie LAN and sent to the PA. The PA then sends an answer back to the ANS part of the vocal interface.

Implementation

The voice component is implemented with a language chosen by the designer, as long as the produced message obeys the OMAS syntax and is broadcast onto the local coterie loop. The answer message must then be picked up from the OMAS messages circulating on the local coterie loop.

On the PA side, processing the message is easy, provided that a specific skill is defined in the PA. For example:

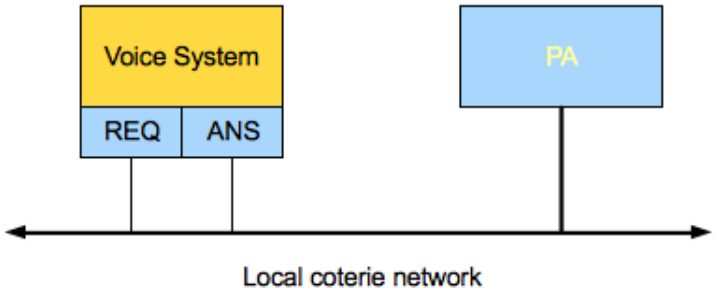

Figure 6.10: Vocal Connection using OMAS messages

```
;;;=============================================================================
;;;                              PROCESS-VOICE-INPUT
;;;============================================================================= skill
;;; receives a label, checks if the category exists. If not, creates it, and sends
;;; a message to the NEWS-PUBLISHER agent.

(defskill :process-voice-input :CIT-STEVENS
  :static-fcn static-process-voice-input
  ;:dynamic-fcn dynamic-process-voice-input
  ;:timeout-handler timeout-handler-process-voice-input
  )

(defun static-process-voice-input (agent message text)
   "function that receives a string from the speech to text program and posts it ~
    into the to-do slot of the PA.
Arguments:
   agent: the PA
   message: the input message
   text: the input text string
Return
   nothing significant."
   (declare (ignore message))
   (format t "~%; static-process-voice-input / text: ~S" text)
   ;; post the received string into the to-do slot of the PA
   (setf (omas::to-do agent) text)
   ;; this will trigger the converse process which presumably is waiting on
   ;; some state
   ;; the output of the converse process should be both a string to be printed into
   ;; the window and sent to the text to speech converter (to do in the dialog part)
   (static-exit agent :done))
```

This skill will send the string to the converse process to be examined by the dialog mechanism.

Returning an answer from the dialog is more difficult and requires to add a special function for broadcasting the answer onto the local coterie net, e.g.:

```
(defun send-voice-answer (text)
   "oribe sends an answer message"
```

```
   (let ((answer (make-instance 'omas::message :from :oribe :to :all :type :inform
            :action :dummy :args (list text))))
     (send-message answer)))
```

In the example the PA is called :CIT-STEVENS.

Variant Another possibility is to specify the :voice parameter in the defassistant macro and to overload the net-voice-send function as follows:

```
;;; clobber the current function
(fmakunbound 'omas::net-voice-send)

;;; redefine OMAS function
(defun omas::net-voice-send (agent text)
  "broadcasts a message using the UDP protocol.
Arguments:
   agent: our PA
   text: a string less than max-message-length bytes long
Return
   code returned by the send-to function."
  ;; check arg
  (unless (stringp text)
    (error "message should be a string rather than: ~S" text))
  (if (> (length text) (omas::voice-max-message-length agent))
      (error "message too long:~%~S" text))

  (let (message)
    ;; trace
    (format t "~%; net-voice-send / sending message to ~A:~S~%;  ~%;   ~S"
      (omas::voice-ip agent) (omas::voice-output-port agent) text)
    ;; make answer message
    (setq message (make-instance 'omas::message
                    :from :CIT-STEVENS :to :all :action :speak :type :answer
                    :args (list text)))
    ;; put the message on the LAN of the local coterie
    (send-message message)
    ))
```

Note that our assistant is :CIT-STEVENS and that the message is sent as a broadcast message. Its type is :answer but it could also be an inform message or any type that the voice component could parse.

Discussion

The method is not recommended for several reasons:

- synthesizing an OMAS message is not difficult but if the OMAS internal syntax of the messages changes, the interface will no longer work.
- parsing the OMAS answer is more tricky, and again assumes that the syntax of the internal messages will not change.
- grabbing the answer obliges to process every message that circulates on the loop.

- processing a request message on the PA side involves a lot of useless overhead, making the program prone to failures.

The approach is thus not recommended.

6.20.3 Using a Postman

The approach here consists of using a postman to receive the input translation of the vocal message and to put it onto the local coterie loop. This way, it can be sent to the PA or broadcast to all the agents on the loop. It can be used if we want to send the vocal input to several agents (or to all agents) of the local coterie. The approach consists of assigning a specific postman (transfer agent) to the vocal component as shown Fig.6.11.

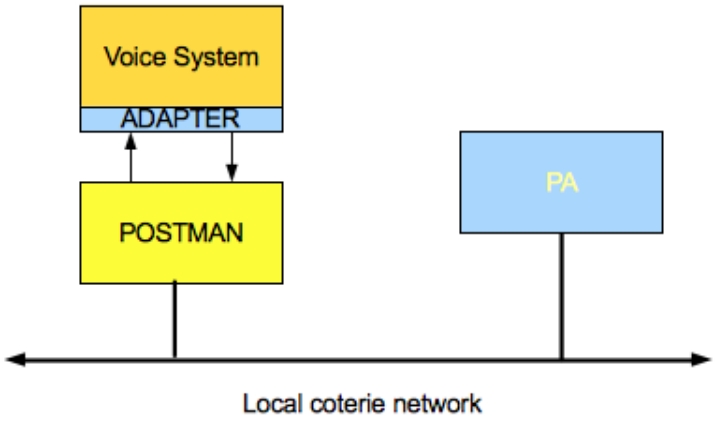

Figure 6.11: Connection to the vocal component using a postman

Implementation

Here the vocal component is connected to the Postman directly using sockets as for the PA direct connection case. The main point is that the postman will take care of formatting the OMAS messages using the right syntax. The postman embodies a proxy agent for the vocal system. If we call the postman :VOICE for example, then we can send and receive messages from the voice component.

The following code is an *untested* example of a possible :VOICE postman.

```
;;;-*- Mode: Lisp; Package: "CIT-VOICE" -*-
;;;============================================================================
;;;12/09/08
;;;                            AGENT POSTMAN :CIT-VOICE
;;; Copyright barthes UTC, 2012
;;; Postman to handle voice input
;;;============================================================================

;;; the postman is only valid for ACL
;;; the voice component is assumed to run on the same machine as the postman
;;; uses ports 900 and 9001
```

```
#|
2012
 0908 creation
|#

(defpackage :CIT-VOICE (:use :moss :omas :cl #+MCL :ccl))
(in-package :CIT-VOICE)

;;;============================================================================
;;;                            Creating postman
;;;============================================================================
;;; the :raw parameter indicates that we will NOT use default skills

(omas::defpostman :CIT-VOICE
    :known-postmen ((:VOICE . "127.1")) :raw t)

;;;============================================================================
;;;                                Globals
;;;============================================================================

(defparameter *voice-input-port* 9000 "port for receiving voice input")
(defparameter *voice-output-port* 9001 "socket for sending string to voice")

(defparameter *voice-input-socket* nil)
(defparameter *voice-output-socket* nil)

(defparameter *voice-receive-process* nil)

(defparameter *max-message-length* 4096)

;;;============================================================================
;;;                             Service functions
;;;============================================================================

(defun voice-receiving (agent)
  "receive a message from the voice input and put it into the to-do slot of the
agent."
  (declare (special *voice-input-socket* *max-message-length*))
  (let (raw-buffer size)
  (loop
    (multiple-value-bind
          (raw-buffer size)
        (socket:receive-from *voice-input-socket* *max-message-length* :extract t)
      (format *debug-io* "+++ ~S ~S" raw-buffer size)
      ;; writes text into the assistant pane of the interface window and puts the
      ;; string into the to-do slot of the agent, reviving the dialog process
      (omas::assistant-process-master-text agent
                                           (make-string-from-buffer raw-buffer size))
      ))))
```

```
;;;------------------------------------------------------------------ VOICE-SEND

(defun voice-send (message)
  "Sends a message to the voice interface using the UDP protocol.
Arguments:
   message: a String less than *max-message-length* bytes long
Return
   code returned by the send-to function."
  ;; check arg
  (unless (stringp message)
    (error "message should be a string rather than: ~S" message))
  (if (> (length message) *max-message-length*)
      (error "message too long:~%~S" message))
  ;; otherwise send message
  (when *voice-output-socket*
    (socket:send-to *voice-output-socket* message (length message)
                    :remote-host "127.1"
                    :remote-port *voice-output-port* agent)))

;;;============================================================================
;;;============================== SKILLS ======================================
;;;============================================================================

;;;============================================================================ skill
;;;                                  CONNECT
;;;============================================================================
;;; the connect skill sets up the sockets for communication with the voice component

(defskill :CONNECT :CIT-VOICE
  :static-fcn static-connect
  )

(defun static-connect (agent message)
  "This function opens communication sockets if necessary.
Arguments:
   agent: assistant agent controlling the vocal interface
Return:
:done"
  (declare (ignore message)
           (special *voice-input-socket* *voice-output-socket*
                    *net-voice-receive-process*))
  ;; create a receiving socket
  (unless *voice-input-socket*
    (setq *voice-input-socket*
          (socket:make-socket
           :type :datagram
           :local-port *voice-input-port*
           ))
    (format t "~%;*** net-initialize-voice / *voice-input-socket* ~S"
      *voice-input-socket*))
```

```
  ;; create a sending socket
  (unless *voice-output-socket*
    (setf *voice-output-socket*
      (socket:make-socket :type :datagram))
    (format t "~%;*** net-initialize-voice / *voice-output-socket* ~S"
      *voice-output-socket*))

  ;; add info to the list of connected agents
  (unless (assoc :voice (omas::connected-postmen-info agent))
    (setf (omas::connected-postmen-info agent)
      (append (omas::connected-postmen-info agent)
              (list (cons :VOICE "127.1")))))

  ;; refresh postman window, which will show connections
  (omas::agent-display (omas::%agent-from-key (omas::key agent)))

  ;; set up receiving process unless it already exists
  (unless *net-voice-receive-process*
    (setq *net-voice-receive-process*
          (mp:process-run-function "Net Voice Receive" #'voice-receiving agent)))
  (static-exit agent :done))


;;;============================================================================ skill
;;;                                DISCONNECT
;;;============================================================================
;;; do we need that? sockets are created with a reuse-address option. If they are
;;; not closed on output, they can be reused when restarting...

(defskill :disconnect :voice
  :static-fcn static-disconnect)

(defun static-disconnect (agent message site-key)
  "closes all sockets and kills receiving process"
  (declare (ignore message))
  (when *voice-input-socket*
    (close *voice-input-socket*)
    (setq *voice-input-socket* nil))
  (when *voice-output-socket*
    (close *voice-output-socket*)
    (setq *voice-output-socket* nil))
  (when *receiving-process*
    (mp:process-kill *receiving-process*)
    (setq *receiving-process* nil))
  ;; must update postman window
  (setf (connected-postmen-info agent)
    (remove site-key (connected-postmen-info agent) :key #'car))
  ;; refresh postman window
  (agent-display (%agent-from-key (key agent)))
```

```
  (static-exit agent :done))

;;;============================================================================ skill
;;;                                  SEND
;;;============================================================================

(defskill :send :voice
  :static-fcn static-send)

(defun static-send (agent in-message message-string message)
  "skill that sees every message and filters those for the voice system,
arguments:
   agent: postman
   in-message: incoming message
   message-string: message to send, a string (ignored)
   message: message to send, an object"
  (declare (ignore in-message message-string))
  (let (text)
    ;; check if message is for voice
    (when (eql :voice (omas::to! message))
      ;; yes transfer the content
      (voice-send (omas::args message)))
    ;; no, get out
    (static-exit agent :done)))

;;; ============================================================================

:EOF
```

Discussion

The postman implementation of the VOICE interface is intended to be used when the voice input could be heard by all agents. It has some overhead but the important point with respect to the previous approach is that all OMAS message formatting or parsing is done by OMAS.

Chapter 7

Transfer Agent or Postman

Contents

Until version 7.14 postmen were defined in a user package and included a number of skills, which turned out to be generic. This OMAS version 7.15 developed a model of generic agent, simplifying the user's programming work.

This chapter contains information about the principle that were chosen to develop the generic postman agent.

7.1 Introduction

7.1.1 Definitions

First we need to recall some definitions.

Agent: An agent is the smallest processing unit.

Coterie Fragment A coterie fragment is a set of agents located on the same machine and executing in the same Lisp environment.

Local Coterie: A local coterie is a set of agents located on a single LAN loop.

Site: A site is a set of loops (local coteries) potentially enclosed by a firewall.

Coterie: A coterie is the set of local coterie residing in the same site.

Application: An application is a set of local coteries that can reside on different sites.

Consequently, we have several kinds of names:

- agent names, e.g. ADDRESS;
- names of files containing several agents executing on the same Lisp environment (i.e. on the same machine), e.g. UTC@DELOS-HDSRI, where UTC@DELOS refers to the machine;
- site names, e.g. UTC;
- application names, e.g. HDSRI (applications may group several coteries located in different sites).

A postman has a name, e.g. UTC, is contained in a specific folder UTC@SKOPELOS-NEWS, belongs to a coterie, on a specific site UTC, for a give application NEWS. When connecting a postman to a remote coterie or to a local coterie, we use the destination postman name and IP.

A given site has a single server (postman) that uses the external site IP address to receive messages from other sites.

Agent names must be unique for a given application.

7.1.2 Principle

A postman is intended to connect OMAS coteries, or an OMAS coterie to a foreign system. Thus, a postman is a gateway.

The default postman is intended for connecting OMAS coteries that can be located on the same site (local coteries) but on different loops, or on different sites (remote coteries).

All messages received by the postman are transferred to the other active coteries automatically, with some exceptions. The exceptions cover:

- system messages (local to a coterie);
- messages specifically addressed to the coterie postman.

7.1.3 Transport Protocol

Starting with version 9.2, there are three possible protocols: direct, client/server or HTTP.

- the direct protocol uses TCP/IP and sockets directly. Each agent is both a server and a client. Messages use receiving port 52008.
- the client serveur protocol is asymmetrical: one postman is the client and the other the server. It uses port 2008 by default.
- the HTTP protocol uses AllegroServe and port 80. Each agent is an AllegroServe server.

The client/server protocol is meant to allow a simple connection for a postman located behind a firewall and that has no external address. Such a postman can initiate a TCP/IP connection and receive messages from a postman that is a known server and that can be reached directly (e.g. the UTC server at nat-omas.utc.fr). Such a postman cannot use the direct connection because it cannot be a server on its own, since it cannot be reached behind the firewall.

7.1.4 Functioning

A postman has two functioning modes: sending and receiving.

Sending

When a postman wants to send a message to another coterie, it only needs to know the name or the IP address of the receiving machine. For some sites the IP is known and fixed. The postman creates a socket for sending the message, tags its identity and coterie name and sends the message. The remote site must be connected, meaning that it must have a program listening on the receiving socket.

Receiving

When a postman is created, it starts a specific receiving process that will wait for a connection on port 52008 for direct connection or on port 80 for HTTP protocol. When a message comes in, if it is an old message, it is discarded. If it is a new message, then the message is broadcast onto the local coterie loop and distributed to other active remote coteries that have not received it yet.

7.1.5 Possible Problems

A number of problems can arise:

- looping: e.g., if a message is transmitted to another postman, then to a third postman, then comes back to the original coterie. OMAS takes care of possible looping conditions.

- IPs: Machines in different places are usually protected by a firewall. Thus the IP is not that of the machine hosting the postman, but the external IP for this machine. On the contrary, if postmen link two internal machines, the IPs are the IPs of the local machines. Note that in general an external address will be given to one machine per site (the address of the OMAS server is nat-omas.utc.fr). Thus, the postman located on the OMAS server can address and receive messages to other sites, other local coteries on the same site must connect to the OMAS server to communicate with coteries located on other sites.

- when a machine is connected to the network in DHCP mode the IP is determined dynamically; i.e. is not known a priori.

- port 52008 of postmen machines must be defiltered to allow message transfers.

7.2 Implementation of the Protocols, Common Features

Although the available protocols have different capabilities, they share some common features regarding data structures and how to define a postman.

7.2.1 Postman Description

A postman description is a structure describing a (remote) postman and is used for setting up a connection. It has the following fields:

- **postman key**: key of the target postman to which we can connect.

- **postman name**: symbolic name of the target postman. For a server behind a firewall reachable from outside, this is its external name, e.g. "nat-omas.utc.fr".

- **postman IP**: IP of the target postman. For a server behind a firewall reachable from outside, this is its external IP.

- **protocol**: one of :TCP, :CLIENT, :HTTP

- **site**: the site of the target postman, e.g. :UTC. Most of the time a postman has the same name as the site on which it resides.

- **port**: the port to be used for communication. Default port is 52008 for direct connection and client/server connection, and 80 for HTTP connection. The field is optional.

The postman description must contain either the name or the IP of the target postman. If the IP is known the connection will be done using the IP, it the IP is not known, then the name will be used to recover the IP through the DNS.

Exemples The following postman descriptions illustrate the approach:

```
(:KC "pegasos" nil :TCP :home)
(:UTC "nat-omas.utc.fr" nil :CLIENT :UTC)
(:TECPAR nil "200.183.132.15" :HTTP :TECPAR)
```

7.2.2 Data Structures

A postman uses a number of data structures containing relevant information.

- **list of known postmen**. This is a list of local postmen names expressed as a list of postman descriptions. First initialized when the postman is created, it is used for connecting to remote sites.
- **list of connected postmen**. The list is the list of sites or local coteries that are currently active, i.e. to which one can send a message. A site or coterie is added to the list as a result of executing a connect skill. If this list is empty, it is not necessary to send messages.
- **receiving-process**. contains the id of the process set up for accepting external messages, only used by the TCP connection.
- **local site**. A key specifying the local site, e.g. :UTC or :CIT.
- **site-ip**. The IP of the target postman site (local coterie).
- **send-socket**. The socket object for sending messages, only used by the TCP connection.
- **receive-socket**. The socket for receiving external messages, only used by the TCP connection.
- **ids-of-received-messages**. A list of ids of recently received messages. Used to check if a new incoming message has been already processed locally. The length of the queue is set by irm-max-size.

7.2.3 Creating a Postman

man is created by the defpostman macro or make-postman function. Thy take a number of parameters (options) detailed in Table 7.1.

Examples The following examples illustrate the approach:

```
(defpostman :UTC
   :site :UTC
   :external-name "nat-omas.utc.fr"
   :internal-name "mikonos"
   :known-postmen ((:TECPAR nil "200.183.132.15" :HTTP :TECPAR))
   )

(omas::defpostman :JPB
   :site :home  ; required
   :internal-name "JEAN-PAULBAC4F6"
   :internal-ip "172.26.138.237"
   :known-postmen (;(:UTC "mikonos" "172.17.130.153" :CLIENT :UTC)
                   (:UTC "nat-omas.utc.fr" "195.83.154.22" :CLIENT :UTC)
                   (:KC "pegasos" nil :CLIENT :HOME))
   )
```

7.2.4 Connecting a New Remote or Local Coterie

When one sends a connect message to the postman directly or using the postman window, the :CONNECT skill is fired. The skill calls different functions according to the remote-postman description in order to set up a connection.

Table 7.1: Parameters to the defpostman macro

parameter	meaning
name	a key designating the postman agent, e.g. :UTC
hide (key)	if t the postman will be hidden, i.e. not appear in the graphics window
http (key)	if t the postman will launch an HTTP sever (Allegroserve)
http-port (key)	the port used for HTTP connections (default is 80)
tcp-port (key)	the port for TCP and client/server connections (default 52008)
server (key)	if t a receiving loop will be started immediately (default t)
known-postmen (key)	a list of remote postmen descriptions
receiving-fcn (key)	a function to be used for creating a receiving loop
proxy (key)	the IP of a proxy for HTTP protocol
raw (key)	if true, means that the user will produce the needed skills
site (key)	the site of the current postman (required)
connection-type (key)	?
internal-name (key	the name of the postman when seen from machines on the site inside the firewall
internal-ip (key)	the IP of the postman when seen from machines on the site inside the firewall
external-name (key)	the name of the postman machine when seen from outside the firewall
external-IP (key)	the IP of the postman machine when seen from outside the firewall
Advanced options	
package (key)	the postman package (default is the name of the postman)
language (key)	proxy language (default is *language*
context (key)	the execution version for MOSS objets (default moss::*context* or 0)
version-graph (key)	the configuration graph (default moss::*version-graph* or '((0)))

7.2.5 Receiving a Message

When the postman is created and the :server option is t a special thread with the postman-receiving function is set up. The function creates a passive socket, then enters a loop creating a stream and connection socket. It then waits for incoming messages on the receiving socket.

If the server has also an HTTP loop then AllegroServe is called. It creates several threads waiting for connections.

In all cases, if the message must be processed a common processing function is called. It does the following:

1. The message string is first converted to an OMAS message object. If this fails, the function quits.
2. The id of the message is then compared to the ids of the messages that have already been received. If the message has already been received it is discarded and the function quits.
3. Otherwise, the id is added to the list of received message ids.
4. If the message is for the receiving postman, it is time stamped and put it into its mailbox. The function quits.
5. Unless the protocol is :CLIENT, the name and IP of the sending postman are extracted from the message.
6. The message is then sent to all connected postmen that have not yet received it.
7. Then, the message is put on the loop of the local coterie.

7.2.6 Sending a Message

When a message appears on the local loop then the postman :SEND skill is activated Sending a message is done either by the default skill or by a :SEND skill that has been defined by the user. The OMAS postman-send function (default skill) does the following:

1. It computes the list of target postmen to which the message is to be sent by removing from the active list itself and the postmen of the thru list contained in the message.
2. If nothing remains, or if the message receiver is the local user, or if the receiver is itself, or if the message is a system message, it does not send the message.
3. Otherwise, it updates the thru list adding itself and the target postmen; it adds an id to the message if there is none, and sends to each target postman.
4. The message to send is converted into a coded string.
5. If an error is detected, the target postman is removed from the list of active postmen and the local postman window is updated.
6. Otherwise the message is sent according to the protocol of the connection.

7.2.7 Disconnecting a Coterie

The :DISCONNECT skill is called either by a message or as the result of clicking the DISCONNECT button in the postman window. It simply terminates the connection, closing sockets and eventually killing listening processes. It then removes the postman description from the list of known connected postmen.

7.2.8 User Point of View

To connect a remote coterie one can use the postman window (Fig.7.1), select the target remote postman and click the connect button.

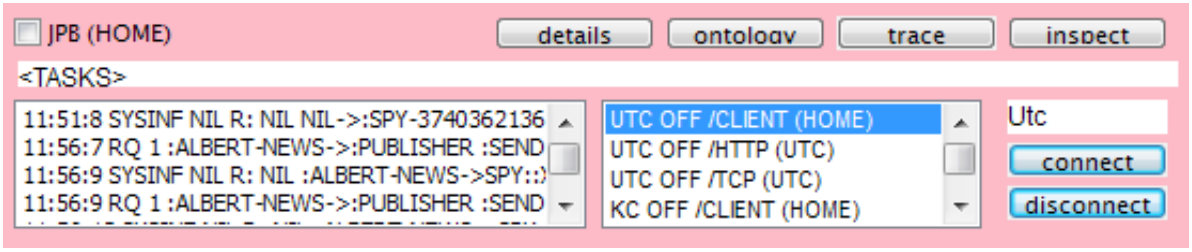

Figure 7.1: Postman Window

If the connection succeeds a green color appears briefly in the list of remote postmen, and the OFF tag switches to ON. If an error occurs, meaning that the remote site is not reachable, a red color appears in the postmen pane and the OFF flag does not change (this may take 30 seconds or more on timeout errors).

Once the remote coterie is connected, it will receive all messages exchanged locally and send all messages exchanged in the remote coterie. Nothing more needs to be done.

If for some reason the remote coterie becomes disconnected, then the ON flag switches to OFF in the postman window.

The disconnect button of the postman window disconnects the associated coterie, and no more messages are sent to it, although messages can still be received. Currently, this is not very useful.

7.2.9 Possible Problems

Several problems may occur:

- A connection to a remote site cannot be achieved. This is usually detected when an :HELLO broadcast does not show any remote answer.
  - the remote site is not listening on port 52008. Ask the site master to launch the receiving process.
  - the name of the remote machine or its IP are not correct. This can happen if the remote machine is protected by a firewall and has an external IP. The external IP must be used.
  - it also may be that the remote site has no agent besides the postman.
- a message appears in the Lisp console saying that the port 52008 is already in use.
  - there has been a crash with a program using this particular port and the corresponding socket was not closed. The simplest solution is to restart Windows.
- Several messages appear in the graphics window, some of which are duplicated.
  - check the names of the remote sites. The name specified in the defpostman parameters must be the same as the name of the remote postman.

7.3 Particulars of the Direct Protocol

In this mode of connection between two postmen, each postman is at the same time both a server and a client. Each postman has a loop containing a function waiting on port 52008 (default) and each connection is done by opening an active socket on the port sending the message, then closing the socket. When the server accepts the connection, it opens a stream, receives the message, then closes the stream and goes wait for another connection request. This implementation is robust but fairly inefficient.

7.3.1 Receiving a Message

When the postman is created and the :server option is t a special thread with the postman-receiving function is set up. The function creates a passive socket, then enters a loop waiting for a connection. When a message comes in the following happens:

1. A stream is created.
2. The message (string) is read and the stream is closed.
3. If the message is empty, go wait from the next one.
4. If the message is a request for a :CLIENT connection, then a special client/server receiving loop is started in a separate process, and the function loops waiting for another connection.
5. Otherwise the message is processed by the common processing function and the function loops waiting for another message.

Note that a client/server connection uses the same port. A request for connection from a client uses a special message, allowing to create a special waiting loop in a new thread.

7.3.2 Disconnecting a Coterie

When the :DISCONNECT skill is called either because the DISCONNECT button of the postman window has been clicked or a disconnect message is received, then the following steps are taken:

1. A message is sent to the postman to be disconnected telling it to disconnect on its side.
2. The postman description is removed from the list of connected postmen info.
3. The postman window is refreshed to show the result.

7.3.3 Postman File

The following piece of code shows a minimal postman for connecting a local coterie called UTC to distant coteries in Japan and in Brazil. It uses the OMAS default skills.

```
;;;-*- Mode: Lisp; Package: "UTC" -*-
;;;============================================================================
;;;10/05/01
;;;                            AGENT POSTMAN :UTC
;;;
;;; Postman to transfer messages from :UTC to remote OMAS platforms
;;;============================================================================
```

```
;;; the postman is only valid for ACL
;;; Note that sites protected by a firewall should specify the GLOBAL IP in the
;;; send message, NOT the IP of the physical machine that sends the message.

;;;============================================================================
;;;                          Creating postman
;;;============================================================================

(omas::defpostman :UTC
             :site :UTC
             :internal-name "mikonos"
             :external-name "nat-omas.utc.fr"
             :known-postmen ((:CIT nil "219.166.183.59" :TCP :CIT)
                             (:TECPAR nil "200.183.132.15" :TCP :TECPAR))
             )

;;; use default skills
:EOF
```

7.4 Particulars of a Client/Server Protocol

The protocol is meant to be used by a postman located behind a firewall and that cannot be a server by lack of external address. In that case the postman can connect to one of the publicly reachable OMAS postmen acting as servers and set up a TCP client/server connection.

7.4.1 Connecting

Connecting is done by opening an active socket on the remote machine and setting up a new process with a function waiting on the active socket.

The connection is done as follows:

1. If the postman description does not include IP or remote machine name, declare an error.
2. Otherwise, get the IP of the remote machine. If not possible display failure in the postman window and quit.
3. Try to open an active socket on the remote machine. If it fails warn and quit.
4. Otherwise, add remote postman info to the list of connected postmen info, save the socket reference onto the plist of the remote postman key
5. Send a message with information describing sending postman and element stating that we want a client connection.
6. Set a receiving loop in a new process (thread) and save the process reference on the plist of the remote postman key.

7.4.2 Receiving a Message

Receiving is done by the function created when connecting. It is a loop untll the received message contains and end-of-file (EOF) marker. It is done as follows:

1. Read the incoming message, posting debug information into the Lisp console.

2. If the message contains an EOF marker, close active socket, clean remote postman key and update the postman window;
3. Otherwise, process the message.
4. Loop waiting for the next message.

The function contains an unwind-protect clause that will close the socket in case an error occurs and the process terminates.

7.4.3 Sending a Message

The sending function recovers the socket-id from the plist of the remote postman key and sends the message using this socket. It contains an unwind-protect clause to guard agains errors, e.g. sending a message when the connection has been closed by the server.

7.4.4 Disconnecting a Coterie

Disconnecting is done as follows:

1. The socket id is obtained from the plist of the remote postman key.
2. A message containing an EOF marker is sent to the remote postman.
3. The socket is closed.
4. The receiving process is killed.
5. The remote postman key is cleaned.

7.4.5 Postman File

The example shows how to declare a postman wanting to connect to the CIT server from behind a firewall, e.g. set by a provider.

```
;;;-*- Mode: Lisp; Package: "UTC" -*-
;;;============================================================================
;;;12/10/5
;;;                            AGENT POSTMAN :JPB
;;;
;;; Postman to transfer messages from :HOME to the :CIT server
;;;============================================================================

(omas::defpostman :JPB
             :site :HOME
             :known-postmen ((:CIT nil "219.166.183.59" :CLIENT :CIT))
             )

;;; use default skills
:EOF
```

7.5 Particulars of an HTTP Protocol

This is done by using AllegroServe on all sites. The global approach is the same. However all exchanges are controlled by AllegroServe. In particular several processes are listening on port 80.

7.5.1 Connecting

The connection is handled by the AllegroServe server. Thus it consists of making sure that the server is up and running, eventually starting it, and making sure that the remote postman is listening by opening an active socket onto the remote machine.

7.5.2 Receiving a Message

Receiving is done by AllegroServe and the message is simply processed.

7.5.3 Sending a Message

Sending is done in the same fashion as for the :TCP protocol. The message is sent through AllegroServe to an address computed from the remote IP, e.g. "200.183.132.15:80/omascc". Furthermore if the address is behind the firewall, a proxy should be specified.

7.5.4 Disconnecting a Coterie

Disconnecting a remote coterie is done by sending an omas message to the remote posman asking is to disconnect and removing the postman info from the connected postmen list, and updating the postman window.

7.5.5 Postman File

The only difference with the previous example is in the defpostman :protocol parameter.

```
;;;-*- Mode: Lisp; Package: "UTC" -*-
;;;============================================================================
;;;12/10/5
;;;                            AGENT POSTMAN :JPB
;;;
;;; Postman to transfer messages from :UTC to remote servers using HTTP protocol
;;;============================================================================

(omas::defpostman :UTC
             : site :UTC
             :protocol :HTTP
             :known-postmen ((:CIT nil "219.166.183.59" :HTTP :CIT)
                             (:TECPAR nil "200.183.132.15" :HTTP :CIT))
             )

;;; use default skills
:EOF
```

7.6 Conclusion

Among the three methods of transfer, the safest one is the one using the TCP protocol historically the first one to be developed. It is however quite inefficient. The lighter one is the one using the CLIENT protocol which is both simple and fast. However, it is brittle and can sometimes crash the platform if the sockets are not closed in due time. The last method using AllegroServe is the easiest one to implement but requires making external addresses and sites explicit to decide whether or not use a proxy.

Chapter 8

Inferer Agent

Contents

Until version 7.14 OMAS agents included Service Agents, Personal Assistants, Transfer Agents or Postmen. One problem with such agents is that they have to be coded in Lisp, which is unbearable to some people. Thus, the Logical Agent or Inferer has been developed to allow users writing rules rather than writing Lisp functions.

A preliminary version of the Inferer has been developed implementing a simple forward-chaining agent.

8.1 Example

8.1.1 Problem

Let us examine the classical movies example by Harmon and King [**?**]. We have an agent that must decide how to go to the movies and has the following rules:

- R1 : If d $>$ 5 km, we drive.
- R2 : If d $>$ 1 km and t $<$ 15 minutes, we drive.
- R3 : If d $>$ 1 km and t $>$ 15 minutes, we walk.
- R4 : If we drive and the theatre is in town, then we take a taxi.
- R5 :If we drive and the theatre is not in town, then we take our car.

- R6 : If we walk and the weather is bad, then we take an umbrella.
- R7 : If we walk and the weather is nice, then we stroll along.

The system is order 0.

8.1.2 Implementation

In order to implement the example, we do the following:

1. We declare an agent with name starting with "IA-" for inference agent, e.g. IA-MOVIES.
2. We add it to the *local-coterie-agents* agent list of the agents.lisp file.
3. We create a file named IA-MOVIES.lisp in which we write the rules.

Rule Format The current format for writing rules is quite simple:

```
(defrule R1
  :if (("d" . ">5"))
  :then ("means" . "car"))
(defrule R2
  :if (("d" . ">1") ("t" ."<15"))
  :then ("means" . "car"))
(defrule R3
  :if (("d" . ">1") ("t" . ">15"))
  :then ("means" . "walk"))
(defrule R4
  :if (("means" . "car") ("cinema" . "in town"))
  :then (:action . "taxi"))
(defrule R5
  :if (("means" . "car") ("cinema" . "not in town"))
  :then (:action . "own car"))
(defrule R6
  :if (("means" . "walk") ("weather" . "bad"))
  :then (:action . "walk with umbrella"))
(defrule R7
  :if (("means" . "walk")("weather" . "nice"))
  :then (:action . "stroll"))
(defrule R8
  :if (("d" . "<1"))
  :then ("means" . "walk"))
```

The precise format is not important and can be changed if needed. It could be:

```
[ R2
  :if  d is >1 and t is <15
  :then means is car ]
```

or any other style of format.

8.1.3 Messages

Inferer agents have the following skills:

- :hello, answers to an hello message;
- :infer-ask, sends a fact and ask if there is an answer;
- :infer-tell, sends a fact to be added to the database
- :infer, asks to start an inference;
- :infer-reset, removes all facts from the fact base.

8.1.4 Tests

The example can be loaded and messages sent to the IA-MOVIES agent, by means of the control panel. The result will appear in the control panel. Messages can be pre-loaded from the Z-messages file, e.g.

```
(defmessage :MV-D<1 :to :movies :type :inform
  :action :infer-tell :args ((:data ("d" . "<1"))))
(defmessage :MV-D>1 :to :movies :type :inform
  :action :infer-tell :args ((:data ("d" . ">1"))))
(defmessage :MV-D>5 :to :movies :type :inform
  :action :infer-tell :args ((:data ("d" . ">5"))))
(defmessage :MV-beau :to :movies :type :inform
  :action :infer-tell :args ((:data ("temps" . "beau"))))
(defmessage :MV-mauvais :to :movies :type :inform
  :action :infer-tell :args ((:data ("temps" . "mauvais"))))
(defmessage :MV-enville :to :movies :type :inform
  :action :infer-tell :args ((:data ("cinema" . "en-ville"))))
(defmessage :MV-T<15 :to :movies :type :inform
  :action :infer-tell :args ((:data ("t" . "<15"))))
(defmessage :MV-T>15 :to :movies :type :inform
  :action :infer-tell :args ((:data ("t" . ">15"))))

(defmessage :mv-INF :to :movies :type :request :action :infer)
(defmessage :mv-RST :to :movies :type :request :action :infer-reset)
```

8.2 Improvements

Several types of improvements can be done.

8.2.1 Better Rule Format

Since rules are translated internally into Lisp structure, any format keeping the semantics of the rules could be defined.

8.2.2 Other types of Inferences

Forward chaining is currently implemented. One could easily implement backward chaining, or mixed strategies.

8.2.3 Order of the Rule System

The example has been implemented with an order-0 system. One could add order-0+ and order-1. Order-1 can be implemented by re-using Peter Norvig [**?**] implementation of unification.

8.2.4 External Inference engines

External inference engines could be called, using the foreign function mechanism.

8.2.5 Model Based Reasoning

Using ontologies and knowledge bases, e.g. from MOSS is a more serious problem, but can be also implemented. The main point consists in exploding all the objects, ontology concepts and individuals, into triples, and using a unification algorithm on the resulting set of triples. This can be done, but is much less efficient than using the MOSS query mechanism directly onto the objects without exploding them. However, I plan to offer the possibility of doing it. In that case, the agent can receive facts, accept queries (logical expression with free variables), or simply be asked to start applying the rules to its database. The internal mechanism is unification (Prolog style).

8.3 Appendix A - Content of the Test File

```
;;;=============================================================================
;;;10/03/20
;;;                  RULE BASE FOR AGENT "MOVIES" (file IA-MOVIES.lisp)
;;;
;;;=============================================================================

#|
2010
 0320 creation for testing IA
|#

(defrule R1
  :if (("d" . ">5"))
  :then ("moyen" . "voiture"))
(defrule R2
  :if (("d" . ">1") ("t" ."<15"))
  :then ("moyen" . "voiture"))
(defrule R3
  :if (("d" . ">1") ("t" . ">15"))
  :then ("moyen" . "a-pied"))
(defrule R4
  :if (("moyen" . "voiture") ("cinema" . "en-ville"))
  :then (:action . "taxi"))
(defrule R5
  :if (("moyen" . "voiture") ("cinema" . "pas-en-ville"))
  :then (:action . "voiture-personnelle"))
(defrule R6
  :if (("moyen" . "a-pied") ("temps" . "mauvais"))
  :then (:action . "a-pied-avec-impermeable"))
(defrule R7
```

```
  :if (("moyen" . "a-pied")("temps" . "beau"))
  :then (:action . "promenade"))
(defrule R8
  :if (("d" . "<1"))
  :then ("moyen" . "a-pied"))


:EOF
```

Chapter 9

Communications

Contents

This chapter concerns communications. It has three parts: the first part describes the OMAS agent communication language, comparing it with FIPA/ACL (FIPA ACL Message Structure Specification SC00061G); the second part describes OMAS content language: the last part describes the network interface.

9.1 Agent Communication Language

9.1.1 Message Structure

A FIPA or OMAS ACL contains one or more parameters. If an agent is unable to process one of the parameters, in the FIPA context it can reply with the not-understood message (presumably this applies to the additional user-defined parameters). In the OMAS context, it simply may not answer at all.

The set of OMAS and FIPA parameters in contained in Table 9.1 and described in the subsequent sections.

Table 9.1: FIPA and OMAS list of parameters

FIPA Parameter	OMAS Parameter	Category
-	name	Message description
-	date	Message description
performative	type	Type of communicative act
receiver	to	Participant in communication
sender	from	Participant in communication
reply-to	reply-to	Participant in communication
content	content, action, args	Content of message
language	-	Description of content
encoding	-	Description of content
ontology	-	Description of content
-	error-contents	Description of content
protocol	protocol	Control of conversation
conversation-id	task-id	Control of conversation
reply-with	-	Control of conversation
in-reply-to	task-id	Control of conversation
reply-by	time-limit	Control of conversation
-	acknowledgement	Control of conversation
-	repeat-count	Control of conversation
-	but-for	Control of conversation
-	strategy	Control of conversation
-	sender-ip	System Information
-	sender-site	System Information
-	thru	System Information
-	task-timeout	System Information
-	timeout	System Information
-	id	Message identifier

9.1.2 Message Description

=== Name

Denotes the name of the message.

FIPA/ACL N/A
OMAS/ACL The name is an identification unique within the local site. Can be used to reference message objects

=== Date

Denotes the time at which the message was created.

FIPA/ACL	N/A
OMAS/ACL	The date is the sender's local time when the message is created. However, when a message arrives onto a different platform the value of the date parameter is changed to the local platform time. If we assume that the travel time is negligible, then it is a way o synchronize clocks. This is not the case however when the coterie comprises remote platforms

9.1.3 Type of Communicative Act

=== Performative / Type

Denotes the type of the communicative act of the FIPA/ACL message or the type of message in the OMAS/ACL. It is a required parameter.

FIPA/ACL	performative
OMAS/ACL	type of message

9.1.4 Participant in Communication

=== Sender / From

Denotes the identity of the sender of the message, that is, the name of the agent of the communicative act in FIPA/ACL and OMAS/ACL.

FIPA/ACL	Maybe any single agent-name
OMAS/ACL	The parameter may have the following values: — a (Lisp) keyword, e.g. :ALBERT as the name of an agent — the special keyword :<USER> or :<site-USER> indicating that the message is sent not to an agent but directly to the user. The result will be posted into the OMAS panel. This is used in debugging conditions — NIL or an empty list or the parameter being omitted. Previously indicating the user of the OMAS panel in debugging conditions. Deprecated.

=== Receiver / To

Denotes the identity of the intended recipients of the message.

FIPA/ACL	Maybe any single agent-name or a non empty set of agent names (multicast)
OMAS/ACL	May be the following things: — a (Lisp) keyword, e.g. :ALBERT as the name of an agent — a non empty list of agent names (multicast) — the special keyword :ALL (broadcast) the intended effect is to broadcast to all reachable agents on the local site, whether they are known or not — the special keyword :ALL-AND-ME. The effect is the same as :ALL but includes the sender as a recipient — a special form representing a MOSS query, that will be evaluated in each agent context by the middleware (stub). If the answer is NIL the message will not be delivered to the receiver, otherwise it will. Please refer to Section 9.1.9 for details. — the special keyword :<USER> or :<site-USER> indicating that the message is sent not by an agent but directly by the user through the OMAS panel. This is used in debugging conditions — NIL or an empty list or the parameter being omitted. Previously indicating the user of the OMAS panel in debugging conditions. Deprecated

=== Reply To

Continuation. The result should be sent to the agents qualified by the content of the reply-to parameter. The syntax is the same as that of receiver.

9.1.5 Content of Message

=== Content

Denotes the content of the message, equivalently denotes the object of the action.The meaning of the content of an ACL message is intended to be interpreted by the receiver of the message.

FIPA/ACL	This is particularly relevant for an instance when referring to referential expressions, whose interpretation might be different for the sender and for the receiver; some messages like cancel have an implicit content, especially in cases using the conversation-id or the in-reply-to parameters
OMAS/ACL	Nothing particular

=== Error Content

Contains an expression giving the reason for the error.

FIPA/ACL	N/A
OMAS/ACL	This is intended for an agent that wants to recover from errors. However, it should be part of the content parameter, associated with a special property error content, to let us use the action and data property for locating the content of the faulty message

9.1.6 Description of Content

=== Language

Denotes the language in which the content parameter is expressed (formal language).This field may be omitted if the agent receiving the message can be assumed to know the language of the current expression.

FIPA/ACL	N/A
OMAS/ACL	The content language is an associated list containing the following properties: — action: denotes the type of skill required. It is a keyword. — data: denotes the arguments to be sent to the skill. — language: denotes the language in which the data is expresses in case of a query or a free language sentence, e.g. :MOSS-QL or :FREE :fr indicating free language in French, etc. — pattern: gives a pattern to express the answer to a MOSS query, expressed as a tree of ontological terms in the specified data language (English by default).

=== Encoding

Denotes the specific encoding of the content language expression.

FIPA/ACL	If the encoding is not present, the encoding is specified in the message envelope that encloses the ACL message.
OMAS/ACL	If the encoding is not specified it defaults to UTF-8. terms in the specified data language (English by default).

=== Ontology

Denotes the ontology used to give a meaning to the symbols in the content expression.

FIPA/ACL	The ontology parameter is used in conjunction with the language parameter to support the interpretation of the content expression y the receiving agent. In many situations, the ontology parameter will be commonly understood by the agent community and this message parameter may be omitted.
OMAS/ACL	Each agent has its own ontology. The ontology parameter is this not very useful, except when sending messages to external FIPA compliant agents.

9.1.7 Control of Conversation

=== Acknowledgement

Denotes a request for specific acknowledgement from the receiver that it received the message. Similar to registered mail.

FIPA/ACL	N/A
OMAS/ACL	Used by the contract-net protocol to allocate tasks and inform losers with a single cancel-grant message. Addressing the message could be done with a predicate in the receiver parameter, however, this is a short-hand syntax specific to the cancel-grant performative. The result is an atomic granting of subtasks that can minimize racing conditions.

=== Protocol

Denotes the interaction protocol that the sending agent is employing with this ACL message.

FIPA/ACL	Any ACL message that contains a non null value for the protocol parameter is considered to belong to a conversation and is required to respect the following rules: — The initiator of the protocol must assign a non null value to the conversation-id parameter. — The responses to the message, within the scope of the same interaction protocol should contain the same value for the conversation-id parameter. — The timeout value of the reply-by parameter must decide the latest time by which the sending agent would like to have received the next message in the protocol flow.
OMAS/ACL	Each message is sent within specific task and is stamped with a task-id parameter. Answers to that message contain the task-id. This parameter is controlled by OMAS directly.

=== Repeat Count

FIPA/ACL	N/A
OMAS/ACL	Since OMAS is using a connectionless transport protocol (UDP) the sender cannot be sure that a message has been received when there is no answer. Thus, if the message is repeated after a timeout, the receiver must identified if it is a next version of a message it has already received in order to avoid doing the task again. The process is handled by OMAS internally, meaning that the parameter should not be set by the programmer.

=== Reply With

Denotes an expression that references an earlier action to which the message is a reply.

OMAS/ACL	Uses the task-id parameter instead.

=== Reply By

Denotes a time and/or date expression which indicates the latest time by which the sending agent would like to receive a reply.

FIPA/ACL The time will be expressed according to the sender's view of the time on the sender's platform. The reply message can be identified in several ways: as the next sequential message in an interaction protocol, through the use of the reply-with parameter, through the use of a conversation-id and so forth. The way that the reply message is identified is determined by the agent implementor.

OMAS/ACL The parameter is time-limit. The time however specifies the maximum delay for the execution of the subcontracted task on the receiver's platform. The rationale is that clocks are not usually synchronized and there may be a gap between the clock in the sender's platform and the user's platform. The maximum waiting time of the sender is specified by a timeout parameter, but this does not need to be transmitted to the receiver....

=== Strategy

Denotes the strategy for selecting answers when several answers are expected, e.g. after a broadcast.

FIPA/ACL N/A

OMAS/ACL The strategy parameter may take three values: take-first-answer, take-first-n-answers or collect-answers. In the latter case a specific timeout should be specified. However, this parameter is handled by the middleware. It is of interest only to the sender and should not be part of the message.

9.1.8 System Information

The flowing parameters are intended for the system. Some should be removed from the message.

=== Sender IP

Contains the IP of the sending site.

FIPA/ACL N/A

OMAS/ACL The value of the parameter is only of interest when messages are sent from a given site to another by a postman. An unknown remote site, e.g. from a notebook must give its IP address so that messages can be sent back. Mainly used for debugging.

=== Sender Site

Denotes the site of the sender of the message.

FIPA/ACL N/A

OMAS/ACL The value of the parameter is only of interest when messages are sent from a given site to another by a postman. Mainly used for debugging.

=== Thru

Denotes the list of sites to which the message has already been sent.

FIPA/ACL	N/A
OMAS/ACL	When a coterie is spread over several remote sites, transfer agents propagate messages to those sites in a transparent fashion. The thru parameter is used to avoid infinite loops. It is handled by the middleware.

=== Task Timeout

Denotes the times the sender is willing to wait for bids in a contract-net protocol.

FIPA/ACL	N/A
OMAS/ACL	The task-timeout parameter is handled by the middleware. It is of interest only to the sender and should not be part of the message. To be removed.

=== Timeout

Denotes the time the sender is willing to wait until an answer comes back.

FIPA/ACL	N/A
OMAS/ACL	The value of the parameter is only of interest to the sender, this should not be part of the message. To be removed.

=== Id

Denotes an identifier for a specific message.

FIPA/ACL	N/A
OMAS/ACL	The value of the parameter is only of interest to remote platforms. It is used to check if the message has already been received from another path.

9.1.9 OMAS Conditional Addressing

OMAS allows conditional addressing, by means of MOSS queries[1]. The query acts as a semi-predicate. E.g.

```
(:_cond ("agent" ("eyes" ("eyes" ("color" :is "blue")))))
```

will deliver messages to all agents with *known* blue eyes.

More specifically, the MOSS query will be executed in the agent environment, and if the result is not NIL, the message will be delivered to the agent. The query may contain anything. Two cases may be distinguished:

- The target class is "agent" like in the previous example. In this case the referred properties are those of the agent.
- The target class is not "agent." In that case, the query is applied to the ontology and knowledge base of the agent, thus addressing the beliefs of the agent. E.g.

  ```
  (:_cond ("sky" ("color" ("color" :is "blue"))))
  ```

  will pass the test if the agent believes that the sky is blue.

Of course the query must contain concepts available in the particular ontology of the agent, otherwise the query will fail.

[1] Please refer to the MOSS documentation for the specification of a MOSS query.

9.2 OMAS Content Language

OMAS does not impose any particular content language. However, a simple structured language can be used for simple applications.

9.2.1 Overall Approach

The result of working on several applications lead us to propose a simple content language, and to define some conventions.

Request messages will contain either a structured MOSS query or a list of words in some natural language. They will specify what should be the form of the answer by providing a pattern.

The answer messages will contain a list of answer structured as specified in the request message or a keyword indicating failure and a list of reasons for the failure.

9.2.2 Structure of the Content of a Message

The content of the message is in the args area of the message and is structured as an association list.

9.2.3 Examples

The first example ask for the answer of a person named Barthès:

```
((:data "quelle" "est" "l" "adresse" "de" "barthès" "?"))
```

As it is formulated the answer will be a list of strings each containing the address of a person named Barthès if there are such persons in the knowledge base.

The second example is analogous to the previous one but specifies the shape of the answer:

```
((:data "quelle" "est" "l" "adresse" "de" "barthès")
 (:pattern "personne" ("nom")("prénom")("ville")))
```

The answer will be a list of items of the form:

```
(("personne" ("nom" "Barthès")("prénom" "Jean-Paul")("ville" "Compiègne"))
 ("personne" ("nom" "Barthès" "Barthès-Biesel")("prénom" "Dominique")("ville" "Compiègne")))
```

The third example uses a MOSS query.

```
((:query ("personne" ("nom" :is "Barthès") ("ville" ("ville" ("nom" :is "compiègne"))))))
```

This form implies that we know the structure of the knowledge base in the agent that will receive the message.

The fourth example specifies the language.

```
((:data "what" "is"  "barthès" "s" "address" "?")
 (:language . :EN))
 (:pattern "person" ("name")("first name")("town")))
```

We had assumed in the previous examples that the language was French by default. Here, we specify that we want English, e.g. using a bilingual address specialist.

9.3 Network Interface

9.3.1 Introduction

OMAS is a multi-agent platform written in Lisp. It has been tested in the Macintosh OS X environment (PPC processors) using MCL 5.2 and in the Windows XP and Windows 7 environments using ACL 8.2.

Communication among the agents uses several levels of protocol, in particular:

- the Agent Communication Language presented in the first part of this chapter
- the Agent Content Language presented in the second part of this chapter
- the exchange protocol that delivers the messages through the network. It can be implemented at a high level, e.g. using CORBA, or lower level using TCP/IP or UDP protocols.

This section describes the communication mechanism of the OMAS system, when agents are distributed, namely how the different protocols are implemented.

9.3.2 Overview

Transmission across machines is done using the UDP protocol and sockets.

Globally, a message is a structured object that must be transformed into a string to be exchanged on the net (marshalling).

Sending Messages

The message is eventually broken into pieces if too long [2], and each piece is sent over the net.

Receiving Messages

A message is received by the net-receive function as a string in a buffer.

A special process waiting on the buffer processes strings as they arrive. Three cases may occur:

1. the message is whole;
2. the message is a piece of a new message;
3. the message is a piece of a message for which we already have received fragments.

The corresponding actions are:

1. If the message is whole, it is transformed back into an object (demarshalling) and put into the input mailboxes of the local agents.
2. If the message is a piece of a new message, then a specific process is created and a timer is activated setting a delay during which the other pieces of the message must arrive.
3. If the message is a piece of a message already identified, then the corresponding task is awaken, the piece is added to the previous fragments. When all the fragments have been received, the message is restructured and dispatched to the local agents.

[2] A typical maximal length for a piece of UDP message is 64K bytes

9.3.3 Exchange Process

Activating the Net

The net mechanism is activated by calling the omas-net-initialize function that calls the net-initialize-broadcast function specific to Mac or Windows environments, and creates a new process for receiving the messages. The global variable `*net-dispatch-process*` references this process.

Activation is done by OMAS unless `(omas::net-broadcast omas::*omas*)` is NIL, meaning that all agents are contained in the same environment.

It is however always possible to call the omas-net-initialize function manually if needed.

Sending Messages

Sending a message is a straightforward process and is done automatically by OMAS with the omas-net-send function. The function takes a message object, translates it into a string, then sends fragments no longer than the value of `*udp-max-message-length*` each fragment having a header containing the fragment number (last one is negative), the name of the sender, the name of the message, the repeat count and a piece of the global message. For example, when the maximal length is 40, we obtain something like the following:

```
"1;FAC-3;MM-0;;(1 MM-0 2 :REQUEST 3 32758"
"2;FAC-3;MM-0;;40314 4 FAC-3 5 MUL-2 6 MU"
"3;FAC-3;MM-0;;LTIPLY 7 (5 7) 11 3600 13 "
"-4;FAC-3;MM-0;;:BASIC-PROTOCOL 17 T-0)"
```

The omas-net-send function uses the net-send function that depends on the environment (calling Open Transport for MCL, using sockets for ACL).

Note that each fragment is sent using UDP in broadcast mode. Thus, nothing whatsoever is done to check whether somebody has received the message.

Note also that the broadcast life parameter is set to 1 by default, meaning that the message is sent on the current local network loop and does not pass to adjacent loops.

Receiving Messages

Receiving messages is a slightly more complex process.

Every time a new fragment comes in, the omas-net-dispatch function is called. It extracts the new fragment, and proceeds as follows:

- if the message is whole, then builds a message object out of the message and dispatches it using the omas-net-distribute-message function.
- otherwise
  - if the fragment is the first one of a new message, then creates a new *message task frame* and adds it to the frame,
  - if the fragment is part of a partially received message, then adds it to the corresponding message task frame using the omas-net-add-and-process-fragment function.

  When all fragments have been received, then they are ordered and merged, a new message object is built (demarshalling) and the message is dispatched.

Message Task Structure

The sending and receiving process for the net is a mechanism common to all local agents, involving however a separate process. Thus, we can use a single global IO-frame, shared by all local agents and containing all the information for running the necessary processes.

A message task frame (MTF) is an object having a number of properties

```
MESSAGE TASK FRAME
   from-name             : name of sending agent, e.g. MUL-3 (NIL if user)
   message-name          : message name, e.g. MM-34
   fragment-list         : list of all pieces of a message
   timeout-process       : timeout process setting the delay for receiving fragments
   fragment-count        : number of pieces to receive (as soon as known
```

Deactivating the Net

Before exiting the net is deactivated by calling the omas-net-close function that will in turn call the net-terminate-broadcast function that depends on the particular environment.

9.3.4 Message Format

OMAS agents use a private protocol to communicate. Two cases occur:

- when agents share the same LISP environment, message objects are passed directly to the agents mailboxes;
- when some agents are located on different machines, messages are sent through the net using the UDP protocol.

This section gives the format of the web messages.

Marshalling and Demarshalling

Using a network connection requires to transform message objects into strings for sending them and to build message objects from the received strings. Furthermore, the UDP protocol limits the length of the messages being transmitted, hence we slice the messages into smaller pieces, fragments, numbering them as we do so. Message fragments can be received out of order, thus, we must be careful when rebuilding a message. Another point is that a message can be transmitted several times by OMAS.

Message Format

The message header contains five fields with the following meaning:

- Sequence number: An integer giving the sequence number of a fragment. The first fragment will be numbered 1, the second 2, etc., the last one being numbered negatively.
- Sender Identifier: name of the agent sending the message
- Message Identifier: name of the message
- Repeat count (optional): an integer giving a count when a message is repeated (1 is the first message; 2 is a repeated message; etc.)
- Content: content of the message fragment.

Example

```
"1;FAC-3;MM-0;;(1 MM-0 2 :REQUEST 3 32758"
"2;FAC-3;MM-0;;40314 4 FAC-3 5 MUL-2 6 MU"
"3;FAC-3;MM-0;;LTIPLY 7 (5 7) 11 3600 13 "
"-4;FAC-3;MM-0;;:BASIC-PROTOCOL 17 T-0)"
```

The example shows a message sent as 4 fragments by agent FAC-3. The message is named MM-0 and the repeat count is omitted. The original message is the following:

```
(1 MM-0 2 :REQUEST 3 3275840314 4 FAC-3 5 MUL-2 6 MULTIPLY 7 (5 7) 11 3600
  13 BASIC-PROTOCOL 17 T-0)
```

Note that the original message is an alternated list of pairs <property><value> where properties are represented by integers (to make messages more compact).

Note also that the different parts of the message header are separated by semi-columns.

The total length of each fragment is specified by a global variable: `*udp-max-message-length*` that was set to 40 in our example. However, its default value is 64000.

Code of the Message Properties

To make message more compact each property of the message is replaced by an integer. The correspondence is given Table 9.2.

Table 9.2: Code of message properties

1	OMAS::NAME
2	OMAS::TYPE
3	OMAS::DATE
4	OMAS::FROM
5	OMAS::TO
6	OMAS::ACTION
7	OMAS::ARGS
8	OMAS::CONTENTS
9	OMAS::CONTENT-LANGUAGE
10	OMAS::ERROR-CONTENTS
11	OMAS::TIMEOUT
12	OMAS::TIME-LIMIT
13	OMAS::ACK
14	OMAS::PROTOCOL
15	OMAS::STRATEGY
16	OMAS::BUT-FOR
17	OMAS::TASK-ID
18	OMAS::REPLY-TO
19	OMAS::REPEAT-COUNT
20	OMAS::TASK-TIMEOUT
21	OMAS::SENDER-IP
22	OMAS::SENDER-SITE
23	OMAS::THRU
24	OMAS::ID

Of course the names of the properties are defined by the OMAS ACL

Receiving Buffer

The sire of the receiving internal buffer is governed by the value of the global variable: `(omas::kTransferBufferSize* omas::*omas*)`. In our case its value is 64000.

Inter-Platform Communications

When we want to couple different platforms located on different network loops, then we must use a transfer agent (alias Postman).

When the remote agents are OMAS agents, then there is nothing special to do except to make sure that the global variable is initialized and the communications channels are set.

When the receiving platform is different from OMAS (e.g. JADE), then the structure of the message has to be changed and rebuilt in terms of JADE. The transfer agent will take OMAS messages and try to produce FIPA compliant messages, using FIPA ACL and eventually SL content. The communication channel can then be different and set specifically by the transfer agent. In that case the transfer protocol may also be changed to TCP if necessary.

9.3.5 OMAS Net Interface

The interface between OMAS and the net (within a coterie) is controlled by global variables[3] can be done and a set of functions;

- `*net-broadcast*` controls the behavior of OMAS. When true, then all messages are broadcast on the web and a special process listens to the web for incoming messages. When false, all communications remain local, and the functions in this file are not used. Default is broadcasting.
- `*local-broadcast-address-and-port*` contains the broadcast address of the local loop, e.g., "172.17.255.255"
- `*net-incoming-message-stack*` contains the incoming messages
- initialize-net-broadcast: initializes the connection and creates a receiving process
- net-send: sends a broadcast message
- terminate-net-broadcast: closes the connection and cleans up

Each platform implements those functions:

- ACL uses sockets from the :sock package from the ACL library (Windows XP, Windows 7)
- MCL uses BSD sockets of Darwin, using the Apple CoreFoundation library (MacOSX)

[3]Global variables have been replaced by slots of an object instance of a site. The object is referred to by the global variable *omas* in the "OMAS" package. Thus, accessing the global variable *net-broadcast* is done by (net-broadcast *omas*) in the "OMAS" package and (omas::net-broadcast omas::*omas*) elsewhere.

Chapter 10

API

Contents

This chapter gives a list of functions that can be used in agent skills. When the functions are exported they can be used without prefix otherwise the prefix omas:: is needed.

Important: This chapter needs to be upgraded to reflect version 9.

Warning: OMAS 7 is a complex platform using advanced features of Lisp environments. In particular OMAS agents are multi-threaded, and the OMAS platform uses windows extensively. This led to several versions of the code, due to the differences in the ACL Lisp on Windows and MCL on Macintoshes. The API functions however can be used in both cases.

Most of the content of the document was produced automatically from the function inline documentation using the User Manual code developed by Mark Kantrowitz at CMU (1991).

10.1 Convention

The reader should be aware of a certain number of conventions used in the system. Some are related to names, others are attached to internal obj-ids.

10.1.1 Elementary function names

All primitive functions start with a % sign, thus following the LISP machine LISP convention.

Some elementary functions are very primitive and somehow dangerous to use, they start with a double %% sign.

10.1.2 Package

Some API functions are exported from the OMAS package. Thus, if one includes the command

```
(use-package :omas)
```

in the application code, they can be used without package prefixing.

The internal functions must be used carefully and prefixed by the omas:: package reference.

10.1.3 Agent names

Because each agent has its own package, agents are referred to by keywords rather than symbols. Each agent name must be unique within its platform (set of coteries).

10.1.4 MOSS

The knowledge representation language MOSS is included in the OMAS environment. Thus, MOSS can be used to define ontologies, knowledge bases, or various representations.

See the MOSS documentation for programming help.

10.2 Global Variables

Starting with version 7.9, global parameters have been grouped into and instance of a CLOS class called SITE. The instance is called *omas* and is a global variable. All values can be accessed using the accessor functions and set by doing a setf on the accessor functions. Previously defined global variables are deprecated. The content of the *omas* object can be examined using the inspect function.

The *omas* object gives the state of the OMAS environment. The names of the global parameters (slots) are given for information. It is strongly recommended not to change their value directly. Note that in MCL special variables (globals) are shared by threads, in ACL they are not.

Note: some parameters appear in more than one table.

10.2.1 Global Parameters

Table 10.1 contains a few parameters governing the state of the OMAS local coterie.

Table 10.1: Global parameters for OMAS

Accessor	Default	Meaning
basic-processes	mp:*all-processes* *active-processes*	initial active processes
debugging	t	currently not used
languages	'(":EN")	language being used (?)
local-user	:<user>	name of local user (control panel)
omas-directory	nil	pathname to OMAS directory
omas-verbose	nil	toggles the debugging trace
omas-window	nil	control panel
ontology	nil	ontology for normalizing concepts
screen	nil	used by Microsoft versions
spy-name	:undefined	the name of the invisible SPY agent (handling graphics trace)
site-name	"*unknown*"	name of the site of the local coterie
test-string	"Ceci est un test..."	a string used for various tests
text-window	nil	window to print text information (ACL only)
voice-interaction	nil	voice flag if T we use the vocal interface (ACL only)

10.2.2 Agents

Table 10.2 contains parameters related to agents.

Table 10.2: Global parameters for agents

Accessor	Default	Meaning
application-name	nil	name of the application, e.g. FAC
local-reference	nil	a unique name characterizing the local machine, e.g. THOR
local-agents	nil	the list of agents local to this Lisp environment (local coterie)
names-of-known-agents	nil	the names of agents known to exist (used by the graphics trace)
names-of-agents-to-display	nil	the names of agents to display in the graphic trace

Continued on next page

Accessor	Default	Meaning
site-name	"*unknown*"	name of the site of the local coterie
traced-agents	nil	list of agents that are traced

10.2.3 Messages

Table 10.3 contains parameters related to messages.

Table 10.3: Global parameters for messages

Accessor	Default	Meaning
message-property-dictionary	nil	dictionary built automatically and remaining constant for coding messages during the marshalling/demarshalling processes

10.2.4 Graphics

Parameters described in Table 10.4 apply to the graphics trace display.

Table 10.4: Global parameters for agents

Accessor	Default	Meaning
application-name	nil	name of the application, e.g. FAC
local-reference	nil	a unique name characterizing the local machine, e.g. THOR
local-agents	nil	the list of agents local to this Lisp environment (local coterie)
names-of-known-agents	nil	the names of agents known to exist (used by the graphics trace)
names-of-agents-to-display	nil	the names of agents to display in the graphic trace
site-name	"*unknown*"	name of the site of the local coterie
traced-agents	nil	list of agents that are traced

10.2.5 Programming Control

Table 10.5 contains parameters related to the control of the OMAS platform. Most parameters are modified by the system when options are selected in the functions allowing to send messages or create subtasks. However, the user could change some of the values if needed, e.g. the number of times a message is resent after a timeout.

Table 10.5: Global parameters for programming control

Accessor	Default	Meaning
basic-processes	mp:*all-processes*	Microsoft: if true timeouts are displayed
	active-processes	MCL: id.
contract-net-strategy	:take-first-bid	contract-net strategy, can be :collect-bids
default-broadcast-repeat-count	3	number of retries while broadcasting
default-broadcast-strategy	:take-first-answer	default broadcast strategy
default-call-for-bids-repeat-count	3	number of times a call-for-bids is sent after timeout
default-contract-net-strategy	:take-first-bid	default contract net strategy
edit-message-window-list	nil	list of temporary windows editing messages
max-repeat-count	3	number of times a message is repeated after a time-out
subtask-number	0	used to create subtask ids
task-number	0	used to create task ids
user-messages	nil	list of test messages produced by the user
voice-interaction	nil	if true we use vocal input (Microsoft only)

10.2.6 Timings

Table 10.6 contains parameters related to default timings.

Table 10.6: Global parameters for timing

Accessor	Default	Meaning
default-call-for-bids-timeout-delay	1	time to wait for an answer to a call-for-bids (1s)
default-execution-time	36000	high number for a default execution time (10h)
default-time-limit	3600	default time limit on all tasks (1h)
highest-time-limit	60000000	a little less than 2 yr
security-timeout	3600	unused

10.2.7 Tracing

Table 10.7 contains parameters governing tracing options.

Table 10.7: Global parameters for tracing

Accessor	Default	Meaning
omas-verbose	nil	toggles the debugging trace
traced-agents	nil	for printing conversations (MCL onlly)
trace-messages	t	if true print message text in graphics window

10.2.8 Network Interface

Table 10.8 contains parameters for handling local loop exchanges. Connections with remote coteries, or other systems are handled by transfer agents.

Table 10.8: Global parameters for network interface

Accessor	Default	Meaning
ktransferbuffersize	4096	size of the buffer for receiving messages
local-broadcast-address	nil	local broadcast address, e.g. 172.17.255.255
local-ip-addres	nil	unused
net-broadcast	nil	if nil no message is broadcast on the net
net-dispatch-process	nil	process that handles reassembling pieces of messages and dispatches the resulting message
net-incoming-message-stack	nil	used by the demarshalling process
net-input-task-list	nil	list of task frames corresponding to processes trying to reassemble incoming messages
net-receive-process	nil	process for receiving local coterie messages
net-send-process	nil	process for sending messages to local LAN loop
omas-port	nil	local port for broadcasting on local loop
socket	nil	reference to broadcasting socket (ACL only)
udp-max-message-length	512	maximal length of UDP messages

10.3 Functions

In this section, API functions have been grouped by features.

10.3.1 Agents

```
;;; ANSWERING-AGENT (agent)                                          [FUNCTION]
;;;    gives the id of the agent corresponding to the received answer
;;;    message being processed.
;;;    Arguments:
;;;       agent: agent.
;;;
;;; ASSISTANT? (agent)                                               [FUNCTION]
;;;    check whether an agent is an assistant, in this case it returns t.
;;;    Arguments:
;;;       agent: agent.
;;;
```

```
;;; CREATE-AGENT-NAME (given-name &optional (package *package*))     [FUNCTION]
;;;    create an internal agent name by prefixing the given name with SA_
;;;    One should use the keywords anyway in the application code.
;;;    Argument:
;;;       given-name: e.g. 'MUL-1
;;;    Return:
;;;       SA_MUL-1
;;;
;;; CREATE-ASSISTANT-NAME (given-name                                  [FUNCTION]
;;;                        &optional (package *package*))
;;;    create an internal assistant name by prefixing the given name with
;;;    PA_ One should use the keywords anyway in the application code.
;;;    Argument:
;;;       given-name: e.g. 'MUL-1
;;;    Return:
;;;       PA_MUL-1
;;;
;;; CREATE-POSTMAN-NAME (given-name &optional (package *package*))   [FUNCTION]
;;;    create an internal assistant name by prefixing the given name with
;;;    PA_ One should use the keywords anyway in the application code.
;;;    Argument:
;;;       given-name: e.g. 'UTC
;;;    Return:
;;;       XA_MUL-1
;;;
;;; GET-AGENT-NUMBER-OF-TIMERS "()"                                    [FUNCTION]
;;;    builds an a-list with the number of timers of the agents from the
;;;    local coterie. An agent has currently one timer per task.
;;;    Uses the *local-agents* list.    Used by the display process,
;;;    asynchronous with respect to calls. Arguments:
;;;       none
;;;    Return:
;;;       an a-list, e.g. ((:FAC . 2)(:MUL . 1))
;;;
;;; GET-AGENT-STATES "()"                                              [FUNCTION]
;;;    builds an a-list with the state of the agents from the local coterie.
;;;    An agent is either :busy if it is executing some task, :idle
;;;    otherwise. Uses the *local-agents* list.
;;;    Arguments:
;;;       none
;;;    Return:
;;;       an a-list, e.g. ((:FAC . :BUSY)(:MUL . :IDLE))
;;;
;;; GET-VISIBLE-AGENT-IDS "()"                                         [FUNCTION]
;;;    get the list of ids of the non hidden local agents
;;;    Arguments:
;;;       none
;;;    Return:
;;;       a list of agent ids.
;;;
```

```
;;; LOCAL-AGENT? (agent-key)                                          [FUNCTION]
;;;    test if an agent represented by a keywod is a local agent, i.e., it
;;;    is in the *local-agents* a-list.
;;;    Arguments:
;;;       agent-name: a symbol, presumably an agent name
;;;    Return
;;;       nil or the agent structure
;;;
;;; RECEIVING-AGENT ((message message))                               [METHOD]
;;;    getting the key of the receiver of the message.
;;;    Argument:
;;;       message: shoulc be a message object
;;;    Return:
;;;       the key of the sender or nil if message is not a message.
;;;
;;; RESET-ALL-AGENTS "()"                                             [FUNCTION]
;;;    clear all agents, reseting input and output trays, and emptying
;;;    tasks; also reseting number of tasks to 0; and clock to 0
;;;
;;; SENDING-AGENT ((message message))                                 [METHOD]
;;;    getting the key of the sender of the message.
;;;    Argument:
;;;       message: shoulc be a message object
;;;    Return:
;;;       the key of the sender or nil if message is not a message.
;;;
;;; AGENT-TRACE (agent-ref text &rest args)                           [FUNCTION]
;;;    function used to trace agent's behavior.
;;;    Arguments:
;;;       agent-ref: agent object, agent name or agent-key
;;;       text: text for string format
;;;       args: arguments for the format variables.
;;;
;;; TRACE-AGENT (agent)                                               [FUNCTION]
;;;    set the agent traced property.
;;;    Arguments:
;;;       agent: agent to trace.
```

10.3.2 Agent Memory

The following functions are related to an agent memory shared by all tasks.

```
;;; DEFFACT (agent fact key &key service info-type ontology)           [MACRO]
;;;    saves some data (a fact) as a memory item, associated with a key. When no key
;;;      is provided makes one and return it.
;;;    Arguments:
;;;       agent: agent
;;;       fact: the stuff to be saved with its own structure
;;;       key (key): a keyword to retrieve the data
;;;       service (key): the service that produced the value
```

```
;;;       info-type (key): type of information (user-defined)
;;;       ontology (key): ontology that allows to understand the fact
;;;    Return:
;;;       key, either the one provded or a synthetic one.
;;;
;;; FORGET (agent index)                                          [FUNCTION]
;;;    removes some data from memory.
;;;    Arguments:
;;;       agent: agent
;;;       index: key associated with memory item
;;;    Return:
;;;       irrelevant
:::
;;; FORGET-ALL (agent)                                            [FUNCTION]
;;;    used to wipe-out agent's memory
;;;    Arguments:
;;;       agent: agent.
;;;
;;; REMEMBER (agent fact key &key service info-type ontology)     [FUNCTION]
;;;    saves data as an object in memory an a-list. The key is memorized in
;;;    the tag slot.
;;;    If no key is present (i.e. is nil), one is synthesized.
;;;    Arguments:
;;;       agent: agent
;;;       fact: data to be saved
;;;       key (key): pattern for recovering data (usually a keyword)
;;;       service (key): service that produced the value
;;;       info-type (key): type of information
;;;       ontology (key): ontology for which the terms have a meaning.
;;;    Return
;;;       key under which the fact is saved."
;;;
;;; RECALL (agent index &key field)                               [FUNCTION]
;;;    function called when trying to recover data from memory. Data is
;;;    organized as an a-list. A piece of data is an object with a time-stamp
;;;    and a value.
;;;    Arguments:
;;;       agent: agent
;;;       index: pattern for recovering data
;;;       field (key): field of interest to be returned (date, type, ontology,
;;;                    service, all none is data itself)
;;;    Return:
;;;       data according to the vale of field.
```

10.3.3 Agent Transient Memory

The following functions are related to the part of the agent memory related to a particular task. This memory disappears when the task exits and the corresponding process is killed.

```
;;; ENV-ADD-VALUES (agent values tag)                             [FUNCTION]
```

```
;;;    replace the agent environment with env.
;;;       Must be called from a task process executing the right task.
;;;    Arguments:
;;;       agent: agent
;;;       env: environment part of the agent
;;;       values: list of values to add to existing value
;;;       tag: property
;;;    Return:
;;;       new list of values.
;;;
;;; ENV-GET (agent tag)                                          [FUNCTION]
;;;    gets the value attached to tag from the agent environment.
;;;       Must be called from a task process executing the right task.
;;;    Arguments:
;;;       agent: agent
;;;       env: environment part of the agent
;;;       values: list of values to add to existing value
;;;       tag: property
;;;    Return:
;;;       new list of values.
;;;
;;; ENV-REM-VALUES (agent values tag &key test)                  [FUNCTION]
;;;    removes the list of values in task environment.
;;;       Must be called from a task process executing the right task.
;;;    Arguments:
;;;       agent: agent
;;;       env: environment part of the agent
;;;       values: list of values to be removes
;;;       tag: property
;;;       test (key): fonction for the :test option of remove
;;;    Return:
;;;       new list of values.
;;;
;;; ENV-SET (agent values tag)                                   [FUNCTION]
;;;    replace the specified property and values in task environment.
;;;       Must be called from a task process executing the right task.
;;;    Arguments:
;;;       agent: agent
;;;       env: environment part of the agent
;;;       values: list of values
;;;       tag: property
;;;    Return:
;;;       new list of values.
;;;
```

10.3.4 Skills

```
;;; DYNAMIC-EXIT (agent result &key internal)                    [FUNCTION]
;;;    user-called function that takes the result from a task, builds up a
;;;    message to forward the answer as required and sends it.
```

```
;;;       However, if the task was an internal task no answer is sent back
;;;    (presumably the dynamic part of the skill will have
;;;    processed the result.    The dynamic-exit is called normally by a
;;;    process executing the skill, thus it commits suicide. It
;;;    will not work if called from a different process (e.g.a
;;;    timeout process). It kills the time-limit timer process. Arguments:
;;;       agent: agent
;;;       result: result to send back to the caller or to the continuation.
;;;       internal (key): if t means that we are getting out of n internal
;;;    process, it                    is not necessary to send an
;;;    answer message Return:
;;;       we never return from this function.
;;;       Either the process is killed or we have an error.
;;;
;;; GET-ENVIRONMENT (agent)                                          [FUNCTION]
;;;    get the environment area contained in the task frame representing the
;;;    current    task. If no task is present declares an error. To be
;;;    called from a requested    skill, not an informed one since inform
;;;    executes in the scan process. Arguments:
;;;       agent: agent.
;;;   Deprecated: use ENV-XXX functions
;;;
;;; PURELY-LOCAL-SKILL? (agent skill)                                [FUNCTION]
;;;    checks if a skill operates locally, i.e., does not spawn any subtask.
;;;    This is verified when there is no dynamic part to the
;;;    skills.    Returns nil if agent does not have the skill.
;;;    Arguments:
;;;       agent: agent
;;;       skill: skill to check.
;;;
;;; STATIC-EXIT (agent arg)                                          [FUNCTION]
;;;    amounts to a noop so far
;;;
;;; UPDATE-ENVIRONMENT (agent env)                                   [FUNCTION]
;;;    replace the agent environment with env.
;;;       Must be called from a task process executing the right task.
;;;    Arguments:
;;;       agent: agent
;;;       env: environment part of the agent.
;;; Deprecated: use ENV-XXX functions
;;;
```

10.3.5 Tracing

```
;;; AGENT-TRACE (agent text &rest args)                              [FUNCTION]
;;;    function used to trace agent's behavior.
;;;    Arguments:
;;;    agent: agent object or agent name
;;;    text: text for string format
;;;    args: arguments for the format variables.
```

```
;;;
;;; TEXT-TRACE (&rest ll)                                              [FUNCTION]
;;;    used for tracing agents. Used by ACL to trace into a special text
;;;    trace window.
;;;
;;; TRACE-AGENT (agent)                                                [FUNCTION]
;;;    set the agent traced property.
;;;    Arguments:
;;;    agent: agent to trace.
;;;
;;; UNTRACE-AGENT (agent)                                              [FUNCTION]
;;;    reset the agent traced property.
;;;    Arguments:
;;;       agent: agent to untrace.
```

10.3.6 Tasks

```
;;; ABORT-CURRENT-TASK (agent &key ignore-queues)                      [FUNCTION]
;;;    When in the process of executing a particular task, the agent might
;;;    decide to abandon the task, killing all subtasks.
;;;    This is useful when an agent has a particular skill but does
;;;    not want to answer a request.
;;;    Arguments:
;;;       agent: agent
;;;       ignore-queues: if t does not clean queues (presumably already
;;;    done).
;;;
;;; %ABORT-CANCEL-TASK (agent task-id sender                           [FUNCTION]
;;;                        &key task-frame no-subtasks ignore-queues
;;;                        cancel no-mark)
;;;    code used to abort or to cancel a task. Aborting is the same as
;;;    canceling except that an abort message is returned to the
;;;    agent that asked for the task. Note that incase the answer
;;;    should be sent to continuations (reply-to), it is not clear
;;;    to whom we should send the message. Arguments:
;;;       agent: agent
;;;       task-id: task-number of the task to be aborted
;;;       sender: agent that sent the task
;;;       no-subtasks (key): if true does not cancel subtasks (presumably
;;;    there are none)    ignore-queues (opt): when t does not check
;;;    input-messages, agenda or waiting tasks    no-mark (key): when t
;;;    does not draw a black mark on the agent life Return:
;;;       does not return if called from the task process
;;;       returns the id of the aborted task if called from somewhere else.
;;;
;;; ABORT-TASK (agent task-id sender &key task-frame no-subtasks       [FUNCTION]
;;;             ignore-queues no-mark)
;;;    Abort a task even if it has not started. The input queues are
;;;    cleaned. If the task was started, the structures are
;;;    removed, the processes killed, and the subtasks canceled.
```

```
;;;    Arguments:
;;;       agent: agent
;;;       task-id: task-number of the task to be aborted
;;;       sender: agent that sent the task
;;;       no-subtasks (key): if true does not cancel subtasks (presumably
;;;    there are none)    ignore-queues (opt): when t does not check
;;;    input-messages, agenda or waiting tasks    no-mark (key): when t
;;;    does not draw a black mark on the agent life Return:
;;;       does not return if called from the task process
;;;       returns the id of the aborted task if called from somewhere else.
;;;
;;; CANCEL-ALL-SUBTASKS (agent &optional task-frame)                [FUNCTION]
;;;    cancel all subtasks launched by a particular agent. Does not cancel ~
;;;    the task that spawned the subtasks. Kill timeout processes
;;;    related to the subtasks if active.
;;;       When the optional task-frame argument is not present, then it is
;;;    assumed that the concerned task is that corresponding to
;;;    current process. Arguments:
;;;       agent: agent.
;;;       task-frame (opt): task-frame of the task that spawned the
;;;    subtasks.
;;;
;;; CANCEL-ANSWERING-SUBTASK (agent)                                [FUNCTION]
;;;    cancel subtask corresponding to the received answer message being
;;;    processed.    Should also remove the subtask-frame from the
;;;    subtask-list slot of the task-frame.    Should be called from the
;;;    task process, e.g. while executing a skill. Arguments:
;;;       agent: agent.
;;;
;;; CANCEL-CURRENT-TASK (agent &key no-subtasks task-frame no-mark)  [FUNCTION]
;;;    same function as abort-current-task, but does not sent error message
;;;    to the agent that required the task. The task to be canceled
;;;    is the one corresponding to the process being executed.
;;;     Arguments:
;;;       agent: agent
;;;       no-subtaks: (key) if t, indicates that no sbtasks have been
;;;    spawned and that it is not necessary to send cancel messages
;;;       task-frame (key): corresponding task-frame if known.
;;;       no-mark (key): when t, does not draq a black mark onthe agent
;;;    life line Return:
;;;       does not return
;;;
;;; CANCEL-SUBTASK (agent subtask-frame)                            [FUNCTION]
;;;    Cancel the subtask corresponding to subtask-frame. Any info put into
;;;    the user-managed environment area should be cleaned by the
;;;    user. We clean input-messages, agenda, timeout process if
;;;    any, and send a message to all agents that were executing
;;;    the subtask as taken from the subtask-frame in the subtask
;;;    list of current-task.    Removes the subtask-frame from the list of
;;;    subtasks.    The function is usually called from the process
```

```
;;;    executing the task that spawned the subtask, but could be
;;;    called from a timeout or a time-limit process. Arguments:
;;;       agent: agent
;;;       subtask-frame: subtask-frame of the subtask to be cancelled.
;;;
;;; CANCEL-TASK (agent task-id sender &key task-frame no-subtasks    [FUNCTION]
;;;              no-mark)
;;;    same function as abort-task, but does not sent error message to the ~
;;;    agent that required the task.
;;;     Arguments:
;;;       agent: agent
;;;       task-id : id of the task to be canceled
;;;       sender: agent that sent the task
;;;       task-frame (opt): task-frame if known
;;;       no-subtaks (key): if t, indicates that no subtasks have been
;;;    spawned and that it is not necessary to send cancel
;;;    messages.    no-mark (kkey): when t, no black mark is drawn on agent
;;;    life Return:
;;;       When called from task process, does not return, otherwise returns
;;;    the task-id.
;;;
;;; CREATE-SUBTASK-ID "()"                                           [FUNCTION]
;;;    called from skills to start a new subtask. Simply provides a subtask
;;;    id. This version uses a global counter rather than gentemp
;;;    so that we can reset task ids when debugging. A subtask-id
;;;    is simply a moisitive number. Arguments:
;;;       agent: agent (ignored)
;;;    Return:
;;;       a positive subtask number, e.g. 233
;;;
;;; CREATE-TASK-ID "()"                                              [FUNCTION]
;;;    a task id is simply a negative number. The task counter is updated
;;;      Arguments:
;;;      agent: agent (ignored)
;;;      Return:
;;;      a negative task number (e.g. -231)
;;;
;;; GET-TIME-LIMIT (agent)                                           [FUNCTION]
;;;    obtain the time-limit associated with the executing task. If none,
;;;    then returns most positive integer in the system.
;;;    Arguments:
;;;       agent: agent.
;;;
;;; MASTER-TASK? (agent task-id)                                     [FUNCTION]
;;;    checks if the task with id task-id is one of the master's task.
;;;    Arguments:
;;;       agent: agent
;;;       task-id: id of the task to be checked.
;;;
;;; PENDING-SUBTASKS? (agent)                                        [FUNCTION]
```

```
;;;    return the list of pending tasks. If processing an answer message,
;;;    then the list contains at least 1, the one corresponding to
;;;    the answer being processed.    The function must be called from the
;;;    process executing the task for which subtasks are checked,
;;;    typically from a skill. Arguments:
;;;       agent: agent.
;;;
;;; SLOW (delay)                                                    [FUNCTION]
;;;    put the process in a wait state during delay time.
;;;    Arguments:
;;;       delay: time to wait in second.
;;;
```

10.3.7 Messages

```
;;; %REMOVE-MESSAGE-FROM-QUEUE (property value queue)                 [MACRO]
;;;    macro to simplify the writing when removing messages from queues ~
;;;    queue must be explicit (keyword)
;;;
;;; CREATE-MESSAGE-ID "()"                                          [FUNCTION]
;;;    creates a new message id that will be used for avoiding infinite
;;;    loops or echos. Arguments:
;;;       none
;;;    Return:
;;;       an integer
;;;
;;; SELECT-BEST-BIDS (agent bid-list)                               [FUNCTION]
;;;    select best bids in a list of bids according to job-parameters found
;;;    in :contents. Currently they are: start-time execution-time
;;;    quality.    For now best bid is the earlier job.
;;;    Arguments:
;;;       agent: agent
;;;       bid-list: list of submitted bids (message objects).
;;;
;;; SEND-INFORM (agent &key (action :inform) to args (delay 0))    [FUNCTION]
;;;    prepares a message containing the parameters for issueing an inform
;;;    to another agent. Default action is :INFORM.
;;;       Does not set up a subtask.
;;;    Arguments:
;;;       agent: agent
;;;       action: (key) skill to be invoked (default: INFORM)
;;;       to: (key) receiver
;;;       args: (key) arguments to the inform message
;;;       delay: additional delay in seconds (default 0).
;;;
;;; %SEND-MESSAGE (message &optional (locally? nil))                [FUNCTION]
;;;    does the actual send as follows:
;;;       - if the message is emitted by a local agent and the to-field is
;;;    not :all-and-me      then we remove the from agent from the local
;;;    agent list    We send a copy of the message to all agents left in
```

```
;;;    the local-agent-list.    When (net-broadcast *omas*) is on, we also
;;;    broadcast on the net Argument:
;;;       message: message to send (unchecked)
;;;       locally? (opt): if T do not send through the network (default is
;;;    nil).
;;;
;;; SEND-MESSAGE (message &key (locally? nil))                    [FUNCTION]
;;;    send a message to other agents.
;;;       - If the target is nil then we assume that the agent is the user.
;;;    We currently    print the answer in the lisp listener (debugging
;;;    mode) and show it in the    control panel
;;;       - if the target is :all then we send the message to all local
;;;    agents    - if the target is :all-and-me we do the same as for :all
;;;    including ourselves    - if the target is a list of agents, it must
;;;    be a list of agent names, then we send to each agent of the
;;;    list    - if the target is a single agent name, then we send the
;;;    message to the target.    In practice we send the message to all
;;;    agents, but draw only what was intended    in order to keep a
;;;    legible graph. In addition we note the current state of the
;;;    agents, since posting will be done asynchronously, and thus the agent
;;;    state    cannot be recovered at that time. Arguments:
;;;       message: message to send (either structure or key)
;;;       locally? (opt): if true does not send the message through the
;;;    network Return:
;;;       :message-sent
;;;
;;; SEND-SUBTASK (agent &key to action args (repeat-count 0)      [FUNCTION]
;;;               task-id delay timeout time-limit
;;;               (protocol :basic-protocol)
;;;               (strategy :take-first-answer) ack type)
;;;    prepares a message containing the parameters for issueing a subtask
;;;    to    another agent. Builds a subtask-frame and adds it to the
;;;    subtask-list slot.    Checks the :protocol variable and determines
;;;    the output message accordingly.    Function is executed by a skill
;;;    (usually), i.e. by a task process, but it can    also be called
;;;    from a timeout process (when relaunching a subtask). Arguments
;;;       agent: agent sending the subtask
;;;       to: (key) agent name
;;;       action: (key) name of the skill required
;;;       args: (key) arguments for the skill
;;;       repeat-count: (key) number of times the task has been repeated
;;;    after timeout,     default is 0 (first time around)
;;;       subtask-id: (key) specific id to identify the task. It should be
;;;    unique.      by default it is created by OMAS
;;;       timeout: (key) timeout delay allowed for executing the subtask
;;;    default is none    protocol: (key) protocol for the message, default
;;;    is :basic-protocol.    ack: wants an acknowledgement message
;;;    Return:
;;;       :done
;;;
```

```
;;; SYSTEM-MESSAGE? (message)                                          [FUNCTION]
;;;    test if message is a system message, i.e. type is :sys-XXX.
;;;    Arguments:
;;;       message: to test
;;;    Return:
;;;       message or nil.
;;;
```

10.3.8 Miscellaneous

Time

```
;;; TIME-STRING (time)                                                 [FUNCTION]
;;;    get an integer representing universal time and extracts a string
;;;    giving    hour:minutes:seconds
;;;
;;; DATE-STRING (time)                                                 [FUNCTION]
;;;    get an integer representing universal time and extracts a string
;;;    giving    year:month:day
;;;
;;; DATE-TIME-STRING (time)                                            [FUNCTION]
;;;    get an integer representing universal time and extracts a string
;;;    giving    year/month/day hour:minute:second
```

Reset, Exit,...

```
;;; OMAS "()"                                                          [FUNCTION]
;;;    start function. Set up a special process that creates a control
;;;    panel. If the local coterie is connected to the net
;;;    (*net-broadcast* is true) then initializes net UDP
;;;    interface.
;;;
;;; OMAS-EXIT "()"                                                     [FUNCTION]
;;;    exits from omas, closing net communications and cleaning whatever
;;;    needs to be cleaned.
```

Multilingual Text

```
;;; GET-TEXT (mln)                                                     [FUNCTION]
;;;    Get the string corresponding to *language* from  multilingual string.
;;;    If not    present, get the English one, if not present get a
;;;    random one.
;;;    Argument:
;;;       mln: a multilingual name (deined in MOSS)
;;;    Return:
;;;       2 values: first is a string, second is a language tag
;;;    Error:
;;;       if not an MLN.
```

General Service

```
;;; MAKE-KEYWORD (input)                                               [FUNCTION]
```

```
;;;    takes a symbol or a string in imput and returns a keyword
;;;
;;; PA (pa-key)                                                    [FUNCTION]
;;;    set current package to pa-key
```

Voice Interface (Microsoft Only)

```
;;; SPEAK (text)                                                   [FUNCTION]
;;;    when the voice interface is activated, voice the argument text. The
;;;    text must    not be too long.
;;;    Arguments:
;;;       text: a string
;;;    Return:
;;;       nil
```

10.4 Functions by Alphabetical Order

```
;;;
;;; %REMOVE-MESSAGE-FROM-QUEUE (property value queue)                [MACRO]
;;;    macro to simplify the writing when removing messages from queues ~
;;;    queue must be explicit (keyword)
;;;
;;; ABORT-CURRENT-TASK (agent &key ignore-queues)                    [FUNCTION]
;;;    When in the process of executing a particular task, the agent might
;;;    decide to abandon the task, killing all subtasks.
;;;    This is useful when an agent has a particular skill but does
;;;    not want to answer a request.
;;;    Arguments:
;;;       agent: agent
;;;       ignore-queues: if t does not clean queues (presumably already
;;;    done).
;;;
;;; %ABORT-CANCEL-TASK (agent task-id sender                         [FUNCTION]
;;;                       &key task-frame no-subtasks ignore-queues
;;;                       cancel no-mark)
;;;    code used to abort or to cancel a task. Aborting is the same as
;;;    canceling except that an abort message is returned to the
;;;    agent that asked for the task. Note that incase the answer
;;;    should be sent to continuations (reply-to), it is not clear
;;;    to whom we should send the message. Arguments:
;;;       agent: agent
;;;       task-id: task-number of the task to be aborted
;;;       sender: agent that sent the task
;;;       no-subtasks (key): if true does not cancel subtasks (presumably
;;;    there are none)    ignore-queues (opt): when t does not check
;;;    input-messages, agenda or waiting tasks    no-mark (key): when t
;;;    does not draw a black mark on the agent life Return:
;;;       does not return if called from the task process
;;;       returns the id of the aborted task if called from somewhere else.
;;;
```

```
;;; ABORT-TASK (agent task-id sender &key task-frame no-subtasks     [FUNCTION]
;;;             ignore-queues no-mark)
;;;    Abort a task even if it has not started. The input queues are
;;;    cleaned. If the task was started, the structures are
;;;    removed, the processes killed, and the subtasks canceled.
;;;    Arguments:
;;;       agent: agent
;;;       task-id: task-number of the task to be aborted
;;;       sender: agent that sent the task
;;;       no-subtasks (key): if true does not cancel subtasks (presumably
;;;    there are none)    ignore-queues (opt): when t does not check
;;;    input-messages, agenda or waiting tasks    no-mark (key): when t
;;;    does not draw a black mark on the agent life Return:
;;;       does not return if called from the task process
;;;       returns the id of the aborted task if called from somewhere else.
;;;
;;; ANSWERING-AGENT (agent)                                          [FUNCTION]
;;;    gives the id of the agent corresponding to the received answer
;;;    message being processed.
;;;    Arguments:
;;;       agent: agent.
;;;
;;; ASSISTANT? (agent)                                               [FUNCTION]
;;;    check whether an agent is an assistant, in this case it returns t.
;;;    Arguments:
;;;       agent: agent.
;;;
;;; CANCEL-ALL-SUBTASKS (agent &optional task-frame)                 [FUNCTION]
;;;    cancel all subtasks launched by a particular agent. Does not cancel ~
;;;    the task that spawned the subtasks. Kill timeout processes
;;;    related to the subtasks if active.
;;;       When the optional task-frame argument is not present, then it is
;;;    assumed that the concerned task is that corresponding to
;;;    current process. Arguments:
;;;       agent: agent.
;;;       task-frame (opt): task-frame of the task that spawned the
;;;    subtasks.
;;;
;;; CANCEL-ANSWERING-SUBTASK (agent)                                 [FUNCTION]
;;;    cancel subtask corresponding to the received answer message being
;;;    processed.    Should also remove the subtask-frame from the
;;;    subtask-list slot of the task-frame.    Should be called from the
;;;    task process, e.g. while executing a skill. Arguments:
;;;       agent: agent.
;;;
;;; CANCEL-CURRENT-TASK (agent &key no-subtasks task-frame no-mark)  [FUNCTION]
;;;    same function as abort-current-task, but does not sent error message
;;;    to the agent that required the task. The task to be canceled
;;;    is the one corresponding to the process being executed.
;;;     Arguments:
```

```
;;;       agent: agent
;;;       no-subtaks: (key) if t, indicates that no sbtasks have been
;;;    spawned and that it is not necessary to send cancel messages
;;;       task-frame (key): corresponding task-frame if known.
;;;       no-mark (key): when t, does not draq a black mark onthe agent
;;;    life line Return:
;;;       does not return
;;;
;;; CANCEL-SUBTASK (agent subtask-frame)                          [FUNCTION]
;;;    Cancel the subtask corresponding to subtask-frame. Any info put into
;;;    the user-managed environment area should be cleaned by the
;;;    user. We clean input-messages, agenda, timeout process if
;;;    any, and send a message to all agents that were executing
;;;    the subtask as taken from the subtask-frame in the subtask
;;;    list of current-task.    Removes the subtask-frame from the list of
;;;    subtasks.    The function is usually called from the process
;;;    executing the task that spawned the subtask, but could be
;;;    called from a timeout or a time-limit process. Arguments:
;;;       agent: agent
;;;       subtask-frame: subtask-frame of the subtask to be cancelled.
;;;
;;; CANCEL-TASK (agent task-id sender &key task-frame no-subtasks    [FUNCTION]
;;;              no-mark)
;;;    same function as abort-task, but does not sent error message to the ~
;;;    agent that required the task.
;;;     Arguments:
;;;       agent: agent
;;;       task-id : id of the task to be canceled
;;;       sender: agent that sent the task
;;;       task-frame (opt): task-frame if known
;;;       no-subtaks (key): if t, indicates that no subtasks have been
;;;    spawned and that it is not necessary to send cancel
;;;    messages.    no-mark (kkey): when t, no black mark is drawn on agent
;;;    life Return:
;;;       When called from task process, does not return, otherwise returns
;;;    the task-id.
;;;
;;; CREATE-AGENT-NAME (given-name &optional (package *package*))     [FUNCTION]
;;;    create an internal agent name by prefixing the given name with SA_
;;;    One should use the keywords anyway in the application code.
;;;    Argument:
;;;       given-name: e.g. 'MUL-1
;;;    Return:
;;;       SA_MUL-1
;;;
;;; CREATE-ASSISTANT-NAME (given-name                                 [FUNCTION]
;;;                        &optional (package *package*))
;;;    create an internal assistant name by prefixing the given name with
;;;    PA_ One should use the keywords anyway in the application code.
;;;    Argument:
```

```
;;;       given-name: e.g. 'MUL-1
;;;    Return:
;;;       PA_MUL-1
;;;
;;; CREATE-POSTMAN-NAME (given-name &optional (package *package*))   [FUNCTION]
;;;    create an internal assistant name by prefixing the given name with
;;;    PA_ One should use the keywords anyway in the application code.
;;;    Argument:
;;;       given-name: e.g. 'UTC
;;;    Return:
;;;       XA_MUL-1
;;;
;;; CREATE-MESSAGE-ID "()"                                            [FUNCTION]
;;;    creates a new message id that will be used for avoiding infinite
;;;    loops or echos. Arguments:
;;;       none
;;;    Return:
;;;       an integer
;;;
;;; CREATE-SUBTASK-ID "()"                                            [FUNCTION]
;;;    called from skills to start a new subtask. Simply provides a subtask
;;;    id. This version uses a global counter rather than gentemp
;;;    so that we can reset task ids when debugging. A subtask-id
;;;    is simply a moisitive number. Arguments:
;;;       agent: agent (ignored)
;;;    Return:
;;;       a positive subtask number, e.g. 233
;;;
;;; CREATE-TASK-ID "()"                                               [FUNCTION]
;;;    a task id is simply a negative number. The task counter is updated
;;;      Arguments:
;;;      agent: agent (ignored)
;;;      Return:
;;;      a negative task number (e.g. -231)
;;;
;;; DYNAMIC-EXIT (agent result &key internal)                         [FUNCTION]
;;;    user-called function that takes the result from a task, builds up a
;;;    message to forward the answer as required and sends it.
;;;       However, if the task was an internal task no answer is sent back
;;;    (presumably the dynamic part of the skill will have
;;;    processed the result.    The dynamic-exit is called normally by a
;;;    process executing the skill, thus it commits suicide. It
;;;    will not work if called from a different process (e.g.a
;;;    timeout process). It kills the time-limit timer process. Arguments:
;;;       agent: agent
;;;       result: result to send back to the caller or to the continuation.
;;;       internal (key): if t means that we are getting out of n internal
;;;    process, it                    is not necessary to send an
;;;    answer message Return:
;;;       we never return from this function.
```

```
;;;       Either the process is killed or we have an error.
;;;
;;; ENV-ADD-VALUES (agent values tag)                                [FUNCTION]
;;;    replace the agent environment with env.
;;;       Must be called from a task process executing the right task.
;;;    Arguments:
;;;       agent: agent
;;;       env: environment part of the agent
;;;       values: list of values to add to existing value
;;;       tag: property
;;;    Return:
;;;       new list of values.
;;;
;;; ENV-GET (agent tag)                                              [FUNCTION]
;;;    gets the value attached to tag from the agent environment.
;;;       Must be called from a task process executing the right task.
;;;    Arguments:
;;;       agent: agent
;;;       env: environment part of the agent
;;;       values: list of values to add to existing value
;;;       tag: property
;;;    Return:
;;;       new list of values.
;;;
;;; ENV-REM-VALUES (agent values tag &key test)                      [FUNCTION]
;;;    removes the list of values in task environment.
;;;       Must be called from a task process executing the right task.
;;;    Arguments:
;;;       agent: agent
;;;       env: environment part of the agent
;;;       values: list of values to be removes
;;;       tag: property
;;;       test (key): fonction for the :test option of remove
;;;    Return:
;;;       new list of values.
;;;
;;; ENV-SET (agent values tag)                                       [FUNCTION]
;;;    replace the specified property and values in task environment.
;;;       Must be called from a task process executing the right task.
;;;    Arguments:
;;;       agent: agent
;;;       env: environment part of the agent
;;;       values: list of values
;;;       tag: property
;;;    Return:
;;;       new list of values.
;;;
;;; FORGET-ALL (agent)                                               [FUNCTION]
;;;    used to wipe-out agent's memory
;;;    Arguments:
```

```
;;;       agent: agent.
;;;
;;; GET-AGENT-NUMBER-OF-TIMERS "()"                                    [FUNCTION]
;;;    builds an a-list with the number of timers of the agents from the
;;;    local coterie. An agent has currently one timer per task.
;;;    Uses the *local-agents* list.    Used by the display process,
;;;    asynchronous with respect to calls. Arguments:
;;;       none
;;;    Return:
;;;       an a-list, e.g. ((:FAC . 2)(:MUL . 1))
;;;
;;; GET-AGENT-STATES "()"                                              [FUNCTION]
;;;    builds an a-list with the state of the agents from the local coterie.
;;;    An agent is either :busy if it is executing some task, :idle
;;;    otherwise. Uses the *local-agents* list.
;;;    Arguments:
;;;       none
;;;    Return:
;;;       an a-list, e.g. ((:FAC . :BUSY)(:MUL . :IDLE))
;;;
;;; GET-ENVIRONMENT (agent)                                            [FUNCTION]
;;;    get the environment area contained in the task frame representing the
;;;    current    task. If no task is present declares an error. To be
;;;    called from a requested    skill, not an informed one since inform
;;;    executes in the scan process. Arguments:
;;;       agent: agent.
;;;
;;; GET-TEXT (mln)                                                     [FUNCTION]
;;;    Get the string corresponding to *language* from  multilingual string.
;;;    If not    present, get the English one, if not present get a
;;;    random one. Argument:
;;;       mln: a multilingual name (deined in MOSS)
;;;    Return:
;;;       2 values: first is a string, second is a language tag
;;;    Error:
;;;       if not an MLN.
;;;
;;; GET-TIME-LIMIT (agent)                                             [FUNCTION]
;;;    obtain the time-limit associated with the executing task. If none,
;;;    then returns most positive integer in the system.
;;;    Arguments:
;;;       agent: agent.
;;;
;;; GET-VISIBLE-AGENT-IDS "()"                                         [FUNCTION]
;;;    get the list of ids of the non hidden local agents
;;;    Arguments:
;;;       none
;;;    Return:
;;;       a list of agent ids.
;;;
```

```
;;; LOCAL-AGENT? (agent-key)                                          [FUNCTION]
;;;    test if an agent represented by a keywod is a local agent, i.e., it
;;;    is in the *local-agents* a-list.
;;;    Arguments:
;;;       agent-name: a symbol, presumably an agent name
;;;    Return
;;;       nil or the agent structure
;;;
;;; MAKE-KEYWORD (input)                                              [FUNCTION]
;;;    takes a symbol or a string in imput and returns a keyword
;;;
;;; MASTER-TASK? (agent task-id)                                      [FUNCTION]
;;;    checks if the task with id task-id is one of the master's task.
;;;    Arguments:
;;;       agent: agent
;;;       task-id: id of the task to be checked.
;;;
;;; OMAS "()"                                                         [FUNCTION]
;;;    start function. Set up a special process that creates a control
;;;    panel. If the local coterie is connected to the net
;;;    (*net-broadcast* is true) then initializes net UDP
;;;    interface.
;;;
;;; OMAS-EXIT "()"                                                    [FUNCTION]
;;;    exits from omas, closing net communications and cleaning whatever
;;;    needs to be cleaned.
;;;
;;; PA (pa-key)                                                       [FUNCTION]
;;;
;;; PENDING-SUBTASKS? (agent)                                         [FUNCTION]
;;;    return the list of pending tasks. If processing an answer message,
;;;    then the list contains at least 1, the one corresponding to
;;;    the answer being processed.    The function must be called from the
;;;    process executing the task for which subtasks are checked,
;;;    typically from a skill. Arguments:
;;;       agent: agent.
;;;
;;; PURELY-LOCAL-SKILL? (agent skill)                                 [FUNCTION]
;;;    checks if a skill operates locally, i.e., does not spawn any subtask.
;;;    This is verified when there is no dynamic part to the
;;;    skills.    Returns nil if agent does not have the skill.
;;;    Arguments:
;;;       agent: agent
;;;       skill: skill to check.
;;;
;;; RECEIVING-AGENT ((message message))                               [METHOD]
;;;    getting the key of the receiver of the message.
;;;    Argument:
;;;       message: shoulc be a message object
;;;    Return:
```

```
;;;       the key of the sender or nil if message is not a message.
;;;
;;; RAL "()"                                                          [MACRO]
;;;
;;; RESET-ALL-AGENTS "()"                                             [FUNCTION]
;;;    clear all agents, reseting input and output trays, and emptying
;;;    tasks; also reseting number of tasks to 0; and clock to 0
;;;
;;; SELECT-BEST-BIDS (agent bid-list)                                 [FUNCTION]
;;;    select best bids in a list of bids according to job-parameters found
;;;    in :contents. Currently they are: start-time execution-time
;;;    quality.    For now best bid is the earlier job.
;;;    Arguments:
;;;       agent: agent
;;;       bid-list: list of submitted bids (message objects).
;;;
;;; SEND-INFORM (agent &key (action :inform) to args (delay 0))       [FUNCTION]
;;;    prepares a message containing the parameters for issueing an inform
;;;    to another agent. Default action is :INFORM.
;;;       Does not set up a subtask.
;;;    Arguments:
;;;       agent: agent
;;;       action: (key) skill to be invoked (default: INFORM)
;;;       to: (key) receiver
;;;       args: (key) arguments to the inform message
;;;       delay: additional delay in seconds (default 0).
;;;
;;; %SEND-MESSAGE (message &optional (locally? nil))                  [FUNCTION]
;;;    does the actual send as follows:
;;;       - if the message is emitted by a local agent and the to-field is
;;;    not :all-and-me      then we remove the from agent from the local
;;;    agent list    We send a copy of the message to all agents left in
;;;    the local-agent-list.    When (net-broadcast *omas*) is on, we also
;;;    broadcast on the net Argument:
;;;       message: message to send (unchecked)
;;;       locally? (opt): if T do not send through the network (default is
;;;    nil).
;;;
;;; SEND-MESSAGE (message &key (locally? nil))                        [FUNCTION]
;;;    send a message to other agents.
;;;       - If the target is nil then we assume that the agent is the user.
;;;    We currently    print the answer in the lisp listener (debugging
;;;    mode) and show it in the    control panel
;;;       - if the target is :all then we send the message to all local
;;;    agents    - if the target is :all-and-me we do the same as for :all
;;;    including ourselves    - if the target is a list of agents, it must
;;;    be a list of agent names, then we send to each agent of the
;;;    list    - if the target is a single agent name, then we send the
;;;    message to the target.    In practice we send the message to all
;;;    agents, but draw only what was intended    in order to keep a
```

```
;;;    legible graph. In addition we note the current state of the
;;;    agents, since posting will be done asynchronously, and thus the agent
;;;    state    cannot be recovered at that time. Arguments:
;;;       message: message to send (either structure or key)
;;;       locally? (opt): if true does not send the message through the
;;;    network Return:
;;;       :message-sent
;;;
;;; SEND-SUBTASK (agent &key to action args (repeat-count 0)         [FUNCTION]
;;;               task-id delay timeout time-limit
;;;               (protocol :basic-protocol)
;;;               (strategy :take-first-answer) ack type)
;;;    prepares a message containing the parameters for issueing a subtask
;;;    to    another agent. Builds a subtask-frame and adds it to the
;;;    subtask-list slot.    Checks the :protocol variable and determines
;;;    the output message accordingly.    Function is executed by a skill
;;;    (usually), i.e. by a task process, but it can    also be called
;;;    from a timeout process (when relaunching a subtask). Arguments
;;;       agent: agent sending the subtask
;;;       to: (key) agent name
;;;       action: (key) name of the skill required
;;;       args: (key) arguments for the skill
;;;       repeat-count: (key) number of times the task has been repeated
;;;    after timeout,     default is 0 (first time around)
;;;       subtask-id: (key) specific id to identify the task. It should be
;;;    unique.      by default it is created by OMAS
;;;       timeout: (key) timeout delay allowed for executing the subtask
;;;    default is none    protocol: (key) protocol for the message, default
;;;    is :basic-protocol.    ack: wants an acknowledgement message
;;;    Return:
;;;       :done
;;;
;;; SENDING-AGENT ((message message))                                [METHOD]
;;;    getting the key of the sender of the message.
;;;    Argument:
;;;       message: shoulc be a message object
;;;    Return:
;;;       the key of the sender or nil if message is not a message.
;;;
;;; SPEAK (text)                                                     [FUNCTION]
;;;    when the voice interface is activated, voice the argument text. The
;;;    text must    not be too long.
;;;    Arguments:
;;;       text: a string
;;;    Return:
;;;       nil
;;;
;;; STATIC-EXIT (agent arg)                                          [FUNCTION]
;;;    amounts to a noop so far
;;;
```

```
;;; SLOW (delay)                                                    [FUNCTION]
;;;    put the process in a wait state during delay time.
;;;    Arguments:
;;;       delay: time to wait in second.
;;;
;;; SYSTEM-MESSAGE? (message)                                       [FUNCTION]
;;;    test if message is a system message, i.e. type is :sys-XXX.
;;;    Arguments:
;;;       message: to test
;;;    Return:
;;;       message or nil.
;;;
;;; TIME-STRING (time)                                              [FUNCTION]
;;;    get an integer representing universal time and extracts a string
;;;    giving    hour:minutes:seconds
;;;
;;; DATE-STRING (time)                                              [FUNCTION]
;;;    get an integer representing universal time and extracts a string
;;;    giving    year:month:day
;;;
;;; DATE-TIME-STRING (time)                                         [FUNCTION]
;;;    get an integer representing universal time and extracts a string
;;;    giving    year/month/day hour:minute:second
;;;
;;; AGENT-TRACE (agent-ref text &rest args)                         [FUNCTION]
;;;    function used to trace agent's behavior.
;;;    Arguments:
;;;       agent-ref: agent object, agent name or agent-key
;;;       text: text for string format
;;;       args: arguments for the format variables.
;;;
;;; TRACE-AGENT (agent)                                             [FUNCTION]
;;;    set the agent traced property.
;;;    Arguments:
;;;       agent: agent to trace.
;;;
;;; TEXT-TRACE (&rest ll)                                           [FUNCTION]
;;;    used for tracing agents. Used by ACL to trace into a special text
;;;    trace window.
;;;
;;; UNTRACE-AGENT (agent)                                           [FUNCTION]
;;;    reset the agent traced property.
;;;    Arguments:
;;;       agent: agent to untrace.
;;;
;;; UPDATE-ENVIRONMENT (agent env)                                  [FUNCTION]
;;;    replace the agent environment with env.
;;;       Must be called from a task process executing the right task.
;;;    Arguments:
;;;       agent: agent
```

```
;;;       env: environment part of the agent.
```

Chapter 11

Persistency

Contents

11.1 Introduction

The OMAS (Open Multi-Agent System) platform has been developed over many years to let application designers prototype applications involving cognitive agents easily. It offers several models of agents and a middleware taking care of the traditional agent machinery like sending messages and applying skills. The goal was to develop a tool in which an application could be programmed by adding a minimum of code in a plugin style.

In order to develop long term applications, one needs persistency. Persistency can be achieved in different ways:

- saving everything into a flat file from time to time and reloading the file after restarting an agent;
- saving objects into a relational database like MYSQL;
- saving objects into an object database.

The first solution is valid when the knowledge base of an agent is small. The agent can then have a goal that saves the world from time to time. However, it does not scale up to large knowledge bases in particular when they change slowly.

The second solution requires an (easy to do) interface to the relational database. However, a relational database is a rigid environment that cannot be modified easily. It cannot cope with dynamically changing classes or objects.

The third solution is ideally suited to save an agent ontology and knowledge base as was demonstrated by the MLF object base that led to the commercial MATISSE product. Lisp environments offer the possibility to save objects in an object store (the Wood's persistent store for MCL, AllegroCache for ACL).

We use the MOSS persistent store mechanism, adapted to the OMAS environment.

11.2 Overall Approach

For internal reasons, only service agents can have a persistent store. Personal Assistants have currently to rely on their staff to save information.

Persistency applies to the agent ontology and knowledge base, meaning that anything that the agent wants to keep over time has to be modeled by the ontology and eventually expressed in the knowledge base.

OMAS agents are defined in their own name space (Lisp package). We use a single store for all the agents of an application executing in the same Lisp environment. The database is partitioned in different spaces, one space per agent. The name of a given partition is the name of an agent.

11.2.1 Declaring Persistency

Persistency is declared when the agent is created, e.g.

```
(defagent :test :persistency t)
```

There is nothing more to do.

11.2.2 Behind the Scene

Declaring persistency has the following effect:

- the first time around:
  - the ontology and knowledge base are read from the agent ontology file, e.g. TEST-ONTOLOGY.lisp.
  - If the database associated with the application does not exist it is created
  - a partition corresponding to the agent name (actually agent key) is created
  - the partition is filled with the concepts, individuals, functions, methods, macros, variables, used by the agent ontology and knowledge base.
  - the agent skills and functions are NOT saved into the database, and will be reloaded at the beginning of each session.
- during a session, objects can be edited in the database and new objects can be created.
- at the end of a session the database is closed.
- at the beginning of a new session, OMAS checks if there is a persistent store that contains information. If so, the ontology file is not loaded, but the database is opened and the partition is connected to the agent space. Objects are loaded on demand.

IMPORTANT Note that once concepts and individuals are stored, if some objects are modified, the object store and the initial text file are no longer compatible (synchronized). The initial ontology file can be used to reinitialize the store, but does not contain any changes that have be done to its content.

11.2.3 Using the Editor

When using the editor to modify objects, two buttons allow terminating an editing session:

- commit: if clicked, all changes will be saved to the database
- abort: if clicked all changes will be ignored and discarded.

11.2.4 Programmed Editing Session

An editing session can be set up by program during which new objects are created and changes are made to other objects. This is done simply by means of three functions (actually agent methods):

- agent-start-changes *agent* starts an editing session.
- agent-abort-changes *agent* abandons the changes and restore the environment to the state preceding the start of the editing session. Disk is not modified.
- agent-commit changes *agent* terminates the editing session and saves everything to disk.

Normally the three functions are all that is needed to handle persistency, since objects are fetched transparently when needed.

11.2.5 Additional Programming Functions

The following functions allow a finer control on the database, but their use is strongly discouraged.

- clear-database *agent*, is used to wipe out the partition corresponding to the agent. The persistent store is not closed afterwards.
- clear-database *agent*, is used to close the persistent store. Used by the system when closing a session.
- load-object *agent key*, reads a key from the persistent store, sets it to the corresponding value and returns the value.
- open-database *agent*, is used to open the persistent store (in case it has been closed or is not yet open); 3 cases:
  - the database exists as well as the agent partition: we open the database and do nothing else
  - the database exists but not the agent partition: we create the partition and send a warning
  - the database does not exist: we create the database, the partition and send a warning
- print-base *agent*, prints the content of the database and values: debugging function, not recommended for casual use.
- store-object *agent key &optional value &key no-commit*, stores a key and value and do a commit unless the :no-commit option is set to t.

Such functions are an interface to the MOSS persistency mechanism. Indeed, because the persistency deals with ontology and knowledge base, it is handled by the MOSS subsystem.

11.3 Implementation

Implementation differs for ACL or MCL.

11.3.1 ACL Implementation

The ACL implementation uses the AllegroCache object store. OMAS creates a folder called MOSS-ONTOLOGIES-ODB in the current application folder. This folder houses the different files implementing the persistent store.

Each persistent agent has a "map" in the store. New maps can be created at any time when other persistent agents join the coterie.

11.3.2 MCL Implementation

The MCL implementation uses the Woods object-store. OMAS creates a file called MOSS-ONTOLOGIES-ODB.odb in the current application folder.

Each persistent agent has a hash table in the root object. New hash tables can be created at any time when other persistent agents join the coterie.

11.3.3 Pathnames

A global variable moss::*database-pathname* contains a pathname either to the folder or to the file according to the environment. In the ACL environment an additional global variable *allegro-cache* duplicates this value.

11.3.4 The Editing Session Mechanism

When starting an editing session (programmed or through the editor mechanism) a special structure called an EDITING BOX is created and assigned to the *editing-box* global variable in the agent package. The editing box has four parts:

- a new-object-ids list that will accept ids of created objects.
- an old-object-values list that will accept values of the modified object prior to their modification.
- a active variable stating that the editing box is active or not
- an owner variable giving the owner of the editing box/

All changes to MOSS objects will be automatically recorded into one of the two lists while the system is in an editing session.

When finished, the agent-abort-changes or the agent-commit-changes will restore or update the object store by setting up a transaction.

Chapter 12

Representation Language

OMAS uses the MOSS representation language. MOSS implements PDM (Property Driven Model). It is a frame-based representation language based upon the idea of default rather than prescription like many languages used to represent ontologies. MOSS allows defining ontologies and is used by the agents in a seamless way, in their ontologies, tasks, or dialogs.

MOSS is described in a number of documents (that will be eventually regrouped into a user's manual) that can be found at:

`http://www.utc.fr/~barthes/MOSS`

Chapter 13

FIPA Compliance

Contents

This chapter explains how an OMAS platform can appear as a FIPA compliant platform.

13.1 Overall Approach

The main idea is that OMAS agents be able to call external agents located on FIPA-compliant platforms and conversely that the OMAS platform appear as a FIPA platform to an external agent. The main problems are to match the agent communication language, making a correspondence between the performatives, processing the content of the messages, simulating a FIPA architecture different from the OMAS P2P approach.

OMAS version 10 is a first attempt to do that by specializing a transfer agent (postman) to take care of FIPA messages.

In this first approach the OMAS platform does not allow registration of external agents. This will be the point of the next version. Thus, here we consider only communications between agents installed on different platforms.

13.2 FIPA Specifications

The following sub-sections summarize some of the points found in the FIPA specifications. The following sections explain how the specifications were implemented in the OMAS platform.

13.2.1 Platform Structure

A FIPA platform has a special structure consisting of agents and services. A detailed discussion of the FIPA platform architecture can be found in the SC00001L document. Here we only consider minimal requirements. Most of the quoted text comes from the SC00023K document.

There are several main concepts:

- agent platform: "An Agent Platform (AP) provides the physical infrastructure in which agents can be deployed. The AP consists of the machine(s), operating system, agent support software, FIPA agent management components (DF, AMS and MTS) and agents."
- service root: "A service-root is a set of service-directory-entries made available to an agent at start-up. This is the mechanism by which an agent can bootstrap lifecycle support services, such as message-transport-services and agent-directory-services, to provide it with a connection to the outside environment."
- agent management system (AMS): "An AMS is a mandatory component of the AP and only one AMS will exist in a single AP. The AMS is responsible for managing the operation of an AP, such as the creation of agents, the deletion of agents and overseeing the migration of agents to and from the AP (if agent mobility is supported by the AP). Since different APs have different capabilities, the AMS can be queried to obtain a description of its AP. [...] The AMS on an AP has a reserved AID of:

```
(agent-identifier
        :name ams@hap_name
        :addresses (sequence hap_transport_address))
```

The name parameter of the AMS (ams@hap_name) is considered to be the Service Root of the AP."

- message transport service: "The Message Transport Service (MTS) delivers messages between agents within an AP and to agents that are resident on other APs. All FIPA agents have access to at least one MTS and only messages addressed to an agent can be sent to the MTS."
- agent-directory service: "A DF is a component of an AP that provides a yellow pages directory service to agents. [...] The DF is an optional component of an AP."

13.2.2 Agent Identity

An agent identity (AID) contains:

- "The name parameter, which is a globally unique identifier that can be used as a unique referring expression of the agent."
- "The addresses parameter, which is a list of transport addresses where a message can be delivered. A transport address is a physical address at which an agent can be contacted."
- "The resolvers parameter, which is a list of name resolution service addresses. Name resolution is a service that is provided by the AMS through the search function."

13.2.3 Agent Communication Language

"An agent-communication-language (ACL) is a language in which communicative acts can be expressed and hence messages constructed."

The ACL gives the structure of the messages according to the different performatives.

13.2.4 Agent Content Language

"Content is that part of a message (where a message is a communicative act) that represents the component of the communication that refers to a domain or topic area. Content is expressed using content-languages. Expressions contained within the content, or the entire content expression itself, can be put into context by one or more ontologies."

13.2.5 Transport Service

"The Message Transport Service (MTS) delivers messages between agents within an AP and to agents that are resident on other APs. All FIPA agents have access to at least one MTS and only messages addressed to an agent can be sent to the MTS."

13.3 A Minimal Implementation

In a minimal implementation, we developed a postman able to send and receive FIPA structured messages using an HTTP transport protocol. To do so, we use the Allegroserve library. In this implementation, we make a correspondence between some of the performatives, the content language is OMAS CL, the OMAS platform does not provide any search service, all OMAS agents have as HTTP address that of the postman.

13.3.1 Sending Messages

In order to send messages, we have to translate an OMAS message into a FIPA message. In the OMAS environment a postman receives all messages. If it knows the identities of the FIPA agents, it can then restructure the OMAS messages and send them to the external addresses.

Sending messages is done by using the NET.ASERVE.CLIENT:DO-HTTP-REQUEST function of Allegroserve.

The postman has a list of all the external agents:

```
(defparameter *fipa-agents*
  '((:jade-fac-1 "jade-fac-1@delos.gi.utc:80/JADE"
                 "http://delos.gi.utc:80/JADE")
))
```

13.3.2 Receiving Messages

When the server is brought up a special entity is created to receive incoming message addressed to http://mikonos.utc:80/acc. A message sent to an OMAS agent at the server's address is caught by the corresponding server entity of Allegroserve. The message is then transformed into an OMAS message and broadcast on the LAN.

13.3.3 Correspondence Between Performatives

Currently, for test purposes only REQUEST and INFORM messages are implemented on the FIPA side and :request, :inform and :answer performatives on the OMAS side.

FIPA Request

A FIPA REQUEST performative translates directly into an OMAS :request performative as follows:

FIPA	OMAS	parameter
sender-AID	from	Agent AID is translated into a keyword using *fipa-agents* list
receiver-AID list	to	Agent AID is translated into a single keyword if the agent is alone or into a list otherwise
content	action	the name qualifying the action (skill) translated into a keyword
	args	the arguments necessary to execute the action
reply-with	task-id	the reply-with tag is transformed into a task-id

FIPA Inform

A FIPA INFORM performative is transformed into an OMAS :inform or an OMAS :answer depending on the content of the in-reply-to field:

FIPA	OMAS	parameter
sender-AID	from	Agent AID is translated into a keyword using *fipa-agents* list
receiver-AID list	to	Agent AID is translated into a single keyword if the agent is alone or into a list otherwise
content	action	the name qualifying the action (skill) translated into a keyword
	args	the arguments necessary to execute the action
	contents	contains the answer to the request
in-reply-to	task-id	the in-reply-to tag is transformed back into the request task-id

OMAS Request

An OMAS request is transformed into a FIPA REQUEST for action message.

OMAS Answer

An OMAS answer is transformed into a FIPA INFORM message.

13.4 Implementation with All Performatives

This section discusses the correspondence between FIPA performatives and OMAS performatives, following the order of the SC00037 FIPA document (Communicative Act Library). OMAS has a smaller set of performatives than FIPA. On the other hand, FIPA performatives suppose the use of the SL language and modal logic. We are mostly interested in matching FIPA and OMAS performatives for simple interactions leaving out modal logic.

In the following sections, the FIPA part reproduces the content of the SC00037 FIPA document.

13.4.1 ACCEPT PROPOSAL

FIPA

The action of accepting a previously submitted proposal to perform an action.

Example:

```
 (accept-proposal
    :sender (agent-identifier :name i)
:receiver (set (agent-identifier :name j))
:in-reply-to bid089
:content
         "((action (agent-identifier :name j) (stream-content movie1234 19))
           (B (agent-identifier :name j) (ready customer78)))"
    :language fipa-sl)
```

OMAS

accept-proposal is an answer.

13.4.2 AGREE

FIPA

The action of agreeing to perform some action, possibly in the future.

Example: Agent i requests j to deliver a box to a certain location; j answers that it agrees to the request but it has low priority.

```
(request :sender (agent-identifier :name i)
         :receiver (set (agent-identifier :name j))
 :content
             "((action (agent-identifier :name j) (deliver box017 (loc 12 19))))"
         :protocol fipa-request
 :language fipa-sl
 :reply-with order567)
(agree :sender (agent-identifier :name j)
       :receiver (set (agent-identifier :name i))
   :content
           "((action (agent-identifier :name j) (deliver box017 (loc 12 19)))
             (priority order567 low))"
    :in-reply-to order567
:protocol fipa-request
:language fipa-sl)
```

OMAS

There is no AGREE performative. The answer to the request will the contain the data giving its meaning. Its translation is an answer message.

13.4.3 CANCEL

FIPA

The action of one agent informing another agent that the first agent no longer has the intention that the second agent perform some action.

Example: Agent j asks i to cancel a previous request-whenever act by quoting the action.

```
(cancel :sender (agent-identifier :name j)
        :receiver (set (agent-identifier :name i))
:content
           "((action (agent-identifier :name j)
     (request-whenever
                :sender (agent-identifier :name j)
:receiver (set(agent-identifier :name i))
:content
                   \"((action (agent-identifier :name i)
      (inform-ref
                         :sender (agent-identifier :name i)
 :receiver (set (agent-identifier :name j))
 :content
```

```
                          \"((iota ?x (=(price widget) ?x))\")
   (> (price widget) 50))"
   ...)))"
        :langage fipa-sl
...)
```

OMAS

This is simply an INFORM message, since the task may not be cancelled. OMAS CANCEL messages correspond to FIPA requests to stop the action...

13.4.4 CALL FOR PROPOSAL (CFP)

FIPA

The action of calling for proposals to perform a given action.

Example: Agent j asks i to submit its proposal to sell 50 boxes of plums.

```
(cfp :sender (agent-identifier :name j)
     :receiver (set (agent-identifier :name i))
 :content
        "((action (agent-identifier :name i) (sell plum 50))
          (any ?x (and (= (price plum) ?x) (< ?x 10))))"
 :ontology fruit-market
 :language fipa-sl)
```

OMAS

this corresponds to an OMAS CALL-FOR-BIDS

13.4.5 CONFIRM

FIPA

The sender informs the receiver that a given proposition is true, where the receiver is known to be uncertain about the proposition.

Example: Agent i confirms to agent j that it is, in fact, true that it is snowing today.

```
(confirm :sender (agent-identifier :name i)
         :receiver (set (agent-identifier :name j))
         :content
              "weather (today, snowing)"
         :language Prolog).
```

OMAS

This is translated into an OMAS INFORM message.

13.4.6 DISCONFIRM

FIPA

The sender informs the receiver that a given proposition is false, where the receiver is known to believe, or believe it likely that, the proposition is true.

Example: Agent i, believing that agent j thinks that a shark is a mammal and attempts to change j's belief.

```
(disconfirm :sender (agent-identifier :name i)
            :receiver (set (agent-identifier :name j))
            :content
               "((mammal shark))"
             :language fipa-sl)
```

OMAS

This is translated into an OMAS INFORM message.

13.4.7 FAILURE

FIPA

The action of telling another agent that an action was attempted but the attempt failed.

Example: Agent j informs i that it has failed to open a file.

```
(failure :sender (agent-identifier :name j)
         :receiver (set (agent-identifier :name i))
         :content
             "((action (agent-identifier :name j) (open \"foo.txt\"))
             (error-message \"No such file: foo.txt\"))"
         :language fipa-sl)
```

OMAS

AN answer to an action with OMAS uses the ANSWER performative. A failure to an action is indicated by a :failure answer associated with the reason for failure.

13.4.8 INFORM

FIPA

The sender informs the receiver that a given proposition is true.

Example: Agent i informs agent j that (it is true that) it is raining today.

```
(inform :sender (agent-identifier :name i)
        :receiver (set (agent-identifier :name j))
        :content
            "weather (today, raining)"
        :language Prolog)
```

OMAS

INFORM messages with OMAS are messages for which no answer is expected. Thus, it may be a statement or a request for action if no answer is expected.

13.4.9 INFORM-IF

FIPA

A macro action for the agent of the action to inform the recipient whether or not a proposition is true.

Example: Agent i requests j to inform it whether Lannion is in Normandy.

```
(request :sender (agent-identifier :name i)
         :receiver (set (agent-identifier :name j))
         :content
             "((action (agent-identifier :name j)
               (inform-if :sender (agent-identifier :name j)
                          :receiver (set (agent-identifier :name i))
                          :content
                               \"in( lannion, normandy)\"
                          :language Prolog)))"
         :language fipa-sl)
```

Agent j replies that it is not.

```
(inform :sender (agent-identifier :name j)
        :receiver (set (agent-identifier :name i))
        :content
            "\+ in (lannion, normandy)"
        :language Prolog)
```

OMAS

This has nothing to do with a message and is a type of action.

13.4.10 INFORM-REF

FIPA

A macro action for sender to inform the receiver the object which corresponds to a descriptor, for example, a name.

Example: Agent i requests j to tell it the current Prime Minister of the United Kingdom.

```
(request :sender (agent-identifier :name i)
         :receiver (set (agent-identifier :name j))
         :content
             "((action (agent-identifier :name j)
               (inform-ref
                     :sender (agent-identifier :name j)
                     :receiver (set (agent-identifier :name i))
                     :content
                          \"((iota ?x (UKPrimeMinister ?x)))\"
```

```
                    :ontology world-politics :
                    :language fipa-sl)))"
          :reply-with query0
          :language fipa-sl)
```

Agent j replies that Tony Blair is the current Prime Minister of the United Kingdom.

```
(inform :sender (agent-identifier :name j)
        :receiver (set (agent-identifier :name i))
        :content
             "((= (iota ?x (UKPrimeMinister ?x)) \"Tony Blair\"))"
        :ontology world-politics
        :in-reply-to query0)
```

Note that a standard abbreviation for the request of inform-ref used in this example is the act query-ref.

OMAS

This has nothing to do with a message and is a type of action.

13.4.11 NOT-UNDERSTOOD

FIPA

The sender of the act (for example, i) informs the receiver (for example, j) that it perceived that j performed some action, but that i did not understand what j just did. A particular common case is that i tells j that i did not understand the message that j has just sent to i.

Example: Agent i did not understand a query-if message because it did not recognize the ontology.

```
(not-understood
     :sender (agent-identifier :name i)
     :receiver (set (agent-identifier :name j))
     :content
        "((action (agent-identifier :name j)
            (query-if
               :sender (agent-identifier :name j)
               :receiver (set (agent-identifier :name i))
               :content
                  \"<fipa-ccl content expression>\"
               :ontology www
               :language fipa-ccl))
            (unknown (ontology \"www\")))"
       :language fipa-sl)
```

OMAS

If an OMAS agent does not understand it simply does not reply. However, NOT-UNDERSTOOD also may indicate an error when performing an action.

13.4.12 PROPAGATE

FIPA

The sender intends that the receiver treat the embedded message as sent directly to the receiver, and wants the receiver to identify the agents denoted by the given descriptor and send the received propagate message to them.

Example: Agent i requests agent j and its federating other brokerage agents to do brokering video-on- demand server agent to get "SF" programs.

```
(propagate
    :sender (agent-identifier :name i)
    :receiver (set (agent-identifier :name j))
    :content
      "((any ?x (registered
        (agent-description
           :name ?x
           :services (set
               (service-description
                  :name agent-brokerage))))
       (action (agent-identifier :name i)
       (proxy
           :sender (agent-identifier :name i)
           :receiver (set (agent-identifier :name j))
           :content
              \"((all ?y (registered
                 (agent-description
                     :name ?y
                     :services (set
                          (service-description :
                               name video-on-demand)))))
                 (action (agent-identifier :name j)
                (request
                    :sender (agent-identifier :name j)
                    :content
                       \"((action ?z5
                            (send-program (category "SF"))))\"
                    :ontology vod-server-ontology
                    :protocol fipa-reqest ...))
                 true)\"
              :ontology brokerage-agent-ontology
              :conversation-id vod-brokering-2
              :protocol fipa-brokering ...))
        (< (hop-count) 5))"
      :ontology brokerage-agent-ontology
       ...)
```

OMAS

Cannot be done with OMAS. The nearest mechanism is that of a postman.

13.4.13 PROPOSE

FIPA

The action of submitting a proposal to perform a certain action, given certain preconditions.

Example: Agent j proposes to i to sell 50 boxes of plums for $5 (this example continues the example of cfp).

```
(propose
   :sender (agent-identifier :name j)
   :receiver (set (agent-identifier :name i))
   :content
      "((action j (sell plum 50))
        (= (any ?x (and (= (price plum) ?x) (< ?x 10))) 5)"
   :ontology fruit-market
   :in-reply-to proposal2
   :language fipa-sl)
```

OMAS

This is either a request, an inform or an answer. It would be better treated as an inform message.

13.4.14 PROXY

FIPA

The sender wants the receiver to select target agents denoted by a given description and to send an embedded message to them.

Example: Agent i requests agent j to do recruiting and request a video-on-demand server to send "SF" programs.

```
(proxy
    :sender (agent-identifier :name i)
    :receiver (set (agent-identifier :name j))
    :content
       "((all ?x (registered(agent-description :name ?x
            :services (set
              (service-description
                  :name video-on-demand)))))
          (action (agent-identifier :name j)
             (request :sender (agent-identifier :name j)
                :content
                   \"((action ?y6 (send-program (category \"SF\"))))\"
                :ontology vod-server-ontology
                :language FIPA-SL :protocol fipa-request
                :reply-to (set (agent-identifier :name i))
                :conversation-id request-vod-1)
             true)"
     :language fipa-sl
     :ontology brokerage-agent
```

```
    :protocol fipa-recruiting
    :conversation-id vod-brokering-1
     ...)
```

OMAS

Not really useful with OMAS since we can use a broadcast message, or a postman.

13.4.15 QUERY-IF

FIPA

The action of asking another agent whether or not a given proposition is true.

Example: Agent i asks agent j if j is registered with domain server d1.

```
(query-if
   :sender (agent-identifier :name i)
   :receiver (set (agent-identitfier :name j))
   :content
      "((registered (server d1) (agent j)))"
   :reply-with r09 ...)
```

Agent j replies that it is not.

```
(inform
   :sender (agent-identifier :name j)
   :receiver (set (agent-identifier :name i))
   :content
      "((not (registered (server d1) (agent j))))"
   :in-reply-to r09)
```

OMAS

Use REQUEST and ANSWER.

13.4.16 QUERY-REF

FIPA

The action of asking another agent for the object referred to by a referential expression.

Example: Agent i asks agent j for its available services.

```
(query-ref
   :sender (agent-identinfier :name i)
   :receiver (set (agent-identifier :name j))
    :content
      "((all ?x (available-service j ?x)))"
   ...)
```

Agent j replies that it can reserve trains, planes and automobiles.

```
(inform
   :sender (agent-identifier :name j)
   :receiver (set (agent-identifier :name i))
   :content
      "((= (all ?x (available-service j ?x))
        (set (reserve-ticket train)
            (reserve-ticket plane)
            (reserve automobile))))
         ...)"
```

OMAS

Use REQUEST and ANSWER.

13.4.17 REFUSE

FIPA

The action of refusing to perform a given action, and explaining the reason for the refusal.

Example: Agent j refuses to i reserve a ticket for i, since there are insufficient funds in i's account.

```
(refuse
   :sender (agent-identifier :name j)
   :receiver (set (agent-identifier :name i))
   :content
      "((action (agent-identifier :name j)
        (reserve-ticket LHR MUC 27-sept-97))
        (insufficient-funds ac12345))"
   :language fipa-sl)
```

OMAS

Use REQUEST and ANSWER. When an OMAS agent does not want to execute an action it simply does not answer. It could answer with a failure and a reason form failure.

13.4.18 REJECT-PROPOSAL

FIPA

The action of rejecting a proposal to perform some action during a negotiation.

Example: Agent i informs j that it rejects an offer from j to sell.

```
(reject-proposal
   :sender (agent-identifier :name i)
   :receiver (set (agent-identifier :name j))
   :content
     "((action (agent-identifier :name j)
       (sell plum 50))
       (cost 200)
       (price-too-high 50))"
   :in-reply-to proposal13)
```

OMAS

Use either INFORM or an ANSWER to a REQUEST.

13.4.19 REQUEST

FIPA

The sender requests the receiver to perform some action. One important class of uses of the request act is to request the receiver to perform another communicative act.

Example: Agent i requests j to open a file.

```
(request
   :sender (agent-identifier :name i)
   :receiver (set (agent-identifier :name j))
   :content
      "open \"db.txt\" for input"
   :language vb)
```

OMAS

This corresponds to an OMAS REQUEST.

13.4.20 REQUEST-WHEN

FIPA

The sender wants the receiver to perform some action when some given proposition becomes true.

Example: Agent i tells agent j to notify it as soon as an alarm occurs.

```
(request-when
   :sender (agent-identifier :name i)
   :receiver (set (agent-identifier :name j))
   :content
     "((action (agent-identifier :name j)
       (inform
          :sender (agent-identifier :name j)
          :receiver (set (agent-identifier :name i))
          :content
             \"((alarm \"something alarming!\"))\"))
        (Done( alarm )))"
   ...)
```

OMAS

Corresponds to a standard request and a goal.

13.4.21 REQUEST-WHENEVER

FIPA

The sender wants the receiver to perform some action as soon as some proposition becomes true and thereafter each time the proposition becomes true again.

Example: Agent i tells agent j to notify it whenever the price of widgets rises from less than 50 to more than 50.

```
(request-whenever
   :sender (agent-identifier :name i)
   :receiver (set (agent-identifier :name j))
   :content
      "((action (agent-identifier :name j)
        (inform-ref
           :sender (agent-identifier :name j)
           :receiver (set (agent-identifier :name i))
           :content
              \"((iota ?x (= (price widget) ?x)))\"))
        (> (price widget) 50))"
   ...)
```

OMAS

Corresponds to a standard request and a goal.

13.4.22 SUBSCRIBE

FIPA

The act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes.

Example: Agent i wishes to be updated on the exchange rate of Francs to Dollars and makes a subscription agreement with j.

```
(subscribe
   :sender (agent-identifier :name i)
   :receiver (set (agent-identifier :name j))
   :content
      "((iota ?x (= ?x (xch-rate FFR USD))))"
```

OMAS

Corresponds to a request for setting up a goal (application dependent).

13.5 Emulating a FIPA Architecture

This approach intends to emulate a FIPA architecture, i.e. to offer the required services to external agents. However, the services are implemented in the OMAS postman, and make the platform appear as a FIPA platform. This approach is currently under development.

13.5.1 Transport Protocol

Currently, the only available transport protocol is HTTP.

13.5.2 Agent Communication Language

All FIPA performatives are taken into account.

13.5.3 Agent Content Language

ALlowed languages are OMAS CL, SL0, and TATIN CL.

13.5.4 FIPA Services

FIPA services are emulated by the OMAS postman.

13.6 Tests

Tests were conducted using a JADE platform on the delos machine as the FIPA external platform, with the OMAS postman on skopelos.

Chapter 14

OMAS vs JADE

Contents

This chapter describes a comparison between the JADE and the OMAS platforms. It uses the JADE 4.2 platform as available in 2012. The chapter has been reviewed and extended by Márcio Fuckner who also developed and added the WADE section.

14.1 Introduction

14.1.1 Main Purpose of the Comparison

JADE is an interesting test platform since it implements the FIPA standards. As such it is intended to develop web-oriented applications. OMAS on the other hand is a non-standard platform developed for implementing complex intelligent agents. It is therefore interesting to compare the two approaches, starting with a simple problem like Problem 0, and comparing various features.

14.1.2 Problem 0

Problem 0 was proposed to test the basic mechanisms of the different multi-agent platforms. It gives information about how agents are programmed, how messages are sent, and how agents are organized in the platform.

Problem 0 is easy to define. An agent named Factorial offers a service, namely computing factorials when given an integer. However, the agent does not know how to multiply two numbers and must subcontract such multiplications to multiplying agents.

14.1.3 Global Remarks

The global approach in JADE is to develop classes of the agents one wants to use. Such classes are then compiled and the JADE environment instantiates agent classes as needed (e.g., by means of the GUI). An instance of a class in JADE has some goal(s) referred to as *behaviours*, normally added in the agent `setup` method.

Analogously to JADE, an OMAS agent lives until it commits suicide, or somebody kills it. However, the global approach in OMAS is different. Each agent is a separate entity. It is not an instance of a class. An agent has *skills* allowing it to perform services when asked and may also have *goals*,declared separately. This distinction between skills and goals in OMAS is the first substantial difference between the platforms, affecting the agent's design and coding approaches, as can be seen in the next sections.

14.1.4 Content of the Chapter

The document contains several parts, each describing a particular approach to the problem of factorial. It starts with the simplest implementation, considering approaches progressively more complex.

- Section 14.2 describes one of the simplest configurations: we have a single FACTORIAL agent and 2 MULTIPLIER agents. Whenever FACTORIAL requires computing a new factorial, it subcontracts multiply operations to one of the two agents randomly, until the result is obtained.
- Section 14.3 studies the question of message handling, i.e., how messages are selected and processed when received by an agent.
- Section 14.4 studies the problem of registering and discovering services.
- Section 14.5 studies the problem of content language.
- Section 14.6 studies the problem of goals and behaviours.
- Section 14.7 studies the Contract-Net protocol.
- Section 14.8 studies the problem of handling time.
- Section 14.9 studies the problem of concurrency.
- Section 14.10 studies the problem of ontologies and how they are used in each platform.
- Section 14.11 shows an alternative implementation for the Problem 0 using the WADE library.
- Section 14.12 discusses the complexity of different implementations.

14.2 Simple Approach to Problem 0

14.2.1 Overall Approach

In the simplest approach, we develop a single FACTORIAL agent and two MULTIPLY agents. Whenever FACTORIAL requires computing a new factorial, it subcontracts multiply operations to one of the two MULTIPLY agents randomly, until the result is obtained.

The following sections detail the global approach for each platform.

JADE

We create two classes: one for FACTORIAL agents, one for MULTIPLY agents. Each class will contain the proper behaviours. The FACTORIAL agent will take its argument from a request message.

Each class is developed as a Java file and compiled with the JADE library. The resulting class files will be used by the JADE platform. Compilation produces a number of class files, one for each class and its inner classes.

OMAS

We create three separate agents, each with its own behavior. The resulting files are put into an application folder, called UTC-FAC where UTC is the name of the local coterie (local platform), and FAC is the name of the application. Each agent file contains all the required functions for the agent to execute. In addition, the UTC-FAC folder includes a file name *agents.lisp* that contains the list of the agents we want to load, and optionally a file named *z-messages.lisp* containing predefined messages that will be available in the debugging environment.

Compiled files are put into a folder named fasl for ACL (Windows). The folder is also contained in the UTC-FAC folder.

14.2.2 Programming the Multiply Agent

JADE

The Java code for the MULTIPLY agent class (named here MulAgentV0) is shown in Figure 14.1. The class is an extension of the Agent class and contains several parts.

A setup method initializes the agent by adding a specific behaviour named MultiplyServer defined thereafter (lines between 11 and 14). In this example it was not necessary to override the takedown method, which is called when the agent terminates.

The MultiplyServer inner class defines the MULTIPLY agent behaviour as cyclic (lines between 21 and 43). Whenever a message comes in, the agent reads it (myAgent.receive() at line 23) and assumes it contains a string representing two integers separated by a semicolon. It then converts that into two integers, multiply them and returns an answer as a string representing the product, using an INFORM performative.

OMAS

The OMAS Lisp code shown in Figure 14.2 contains several parts.

A first part defines a name space (:mul-1) for the agent, so that each agent can use its own code without fearing to interfere with other functions of other agents (lines 7 and 8) .

The agent is defined by the omas::defagent macro (line 10). Its skills are defined in the skill section by means of the defskill macro. The only skill section specifies a :multiply skill, implemented by the static-multiply function. A static function corresponds to an atomic action. It does not involve subcontracting as will be the case for FACTORIAL.

Comparing JADE and OMAS

On this simple agent several points may be compared:

- *Nature of the Agent File*

  The JADE file defines a class. JADE agents are instances of this class. The OMAS file defines a single agent. Thus OMAS will require another file for the second MULTIPLY agent that may be similar or completely different from the first one.

  Both JADE and OMAS agents are persistent, meaning that they will stay alive until something kills them.

- *Behaviours and skills*

  The JADE agent declares a *behaviour*, the OMAS agent declares a *skill*. The behaviour of the JADE agent needs to be cyclic so that it can serve repeated requests. The skill of the OMAS agent is persistent until explicitly removed.

  JADE behaviours are implemented by classes, OMAS skills are implemented by functions.

- *Processing the Message*

  By default, behaviours in JADE are executed in a single thread[1], configuring a cooperative rather than preemptive multitasking approach. Thus, the program should be slightly different in order to avoid greedy behaviours. For example, the JADE agent explicitly retrieves the

[1] It is possible to run behaviours in a dedicated thread. We will explore this feature in the section 14.9

```
1  // Package and import declarations removed for the sake of clarity
2
3  /**
4   * Agent Multiply receives 2 numbers, multiplies them and returns the result
5   */
6  public class MulAgentV0 extends Agent {
7
8     /**
9      * Agent initialization
10     */
11    protected void setup() {
12       // Add the behaviour serving queries from Fac agents
13       addBehaviour(new MultiplyServer());
14    }
15
16    /**
17     * Inner class MultiplyServer. This is the behaviour used by Mul agents to
18     * serve incoming requests Takes the incoming numbers and return an INFORM
19     * message with the answer.
20     */
21    private class MultiplyServer extends CyclicBehaviour {
22       public void action() {
23          ACLMessage msg = myAgent.receive();
24          if (msg != null) {
25             // Request message received. Process it
26             String argString = msg.getContent();
27             ACLMessage reply = msg.createReply();
28
29             int arg1, arg2;
30             // recover args from string "arg1;arg2"
31             arg1 = Integer.parseInt(argString.substring(0,
32                   argString.indexOf(";")));
33             arg2 = Integer.parseInt(argString.substring(
34                   1 + argString.indexOf(";"), argString.length()));
35
36             reply.setPerformative(ACLMessage.INFORM);
37             reply.setContent(Integer.toString(arg1 * arg2));
38             myAgent.send(reply);
39          } else {
40             block();
41          } // end of message test
42       } // end of action
43    } // end of inner class
44 } // end of class
```

Figure 14.1: Java code for the MULTIPLY agent class

message using the *receive* method. This is a non-blocking method that reads the message queue and returns immediately. As a side effect, a given execution could receive a null message and the code must be prepared to deal with this situation. The source-code shown Figure 14.1 shows a strategy that executes lines from 25 to 38 when a message is successfully retrieved and calls the block method (line 40) when the message queue is empty. In contrast, retrieving messages in OMAS is done automatically in the sense that the skill associated with the message is executed in a separate thread. In addition a timer will kill the thread after one hour by default to prevent unfinished processes to clutter the environment.

```
1  ;;; ============================================================
2  ;;; 04/04/04
3  ;;;                          AGENT MUL-1
4  ;;;
5  ;;; ============================================================
6
7  (defpackage :mul-1 (:use-package :cl :omas :moss))
8  (in-package :mul-1)
9
10 (omas::defagent :mul-1)
11
12 ;;; ============================== skill section ===============================
13
14 (defskill :multiply :mul-1
15   :static-fcn static-multiply)
16
17 (defun static-multiply (agent message n1 n2)
18   (declare (ignore message))
19   (sleep (1+ (random 2))) ; slow down execution so that we can see something
20   (static-exit agent (* n1 n2))) ; return result to caller
```

Figure 14.2: OMAS code for the MULTIPLY agent

The JADE agent extraction process returns any message addressed to the agent[2], thus we assume that the messages are MULTIPLY requests and are well formatted. The OMAS system handles the selection process, calling the particular skill specified in the message, here MULTIPLY. All other messages are ignored.

The JADE agent constructs an explicit INFORM message for returning the answer to the calling agent. The OMAS agent uses the static-exit API function and the answer message is built automatically and returned to the caller.

- *Message Content*

  In the simple approach the JADE agent must decode the content of the message, i.e. the string obtained by [msg.getContent()][3]. The OMAS agent gets the arguments of the message as the last arguments of the skill.

- *Name Space*

  The JADE agent operates by default in its own object space using private variables. The OMAS agent operates in its own package, which requires programmers to know about packages.

14.2.3 Programming the Factorial Agent

The FACTORIAL agent is more complex than the MULTIPLY agent. It deals with end-user requests and also subcontracts tasks to MULTIPLY agents.

JADE

The main part of the Java code for the FAC agent is given in Figure 14.3. The *setup* method creates a cyclic behaviour, similarly to the implementation of the MULTIPLY agent. The agent is composed of four behaviours, which will be explained in separate paragraphs.

[2]We will see later that a message pattern can be used to retrieve only selected types of messages.
[3]We will see that an ontology can be used to transform the content into a Java object.

RequestPerformer behaviour A cyclic behaviour was used in order to maintain the skill active during the agent lifecycle. This behaviour reads messages from the end-users requesting the factorial calculation. As can be seen, line 25, we are using the receive, instead of the blockingReceive method. A blockingReceive call at this point would have a negative impact, preventing other behaviours to execute until a message arrives. One could argue that this agent has only one behaviour and a blocking method will not cause any problems in this scenario. However, much effort would be necessary to adapt the agent to manage multiple behaviours. As a good practice one must program the agent to be capable of running several behaviours in a cooperative way.

After receiving and parsing the request, the behaviour delegates the tasks to more specialized behaviours: If the requested number is less or equal than 2, the behaviour responsible for sending directly the answer is added, preventing unnecessary calls to the MULTIPLY agents. However, if the number is greater than 2, a behaviour responsible for the multiplication request is added. We use the *data store* feature in order to exchange information between behaviours, such as intermediate results and common objects, as can be seen line 35.

MultiplySender behaviour The behaviour responsible for sending the multiplication request is quite simple as shown Figure 14.4. It gets the previous answer from the data store and builds a new message to one of the MULTIPLY agent picked randomly. Finally, a behaviour to retrieve the response is added.

MultiplyReceiver behaviour The behaviour shown Figure 14.5 is responsible for retrieving the MULTIPLY agent result, deciding if it is necessary to subcontract more multiplying agents or finish the task, sending a message back to the user. This is done adding specialized behaviours for each tasks rather than declare everything here.

The level of granularity shown here was not only chosen for the sake of componentization, but to comply with the intrinsic characteristics of the JADE scheduler. Just to remember: By default, behaviours are executed in a single thread and the message reception is done using the *receive* method. Once it is a non-blocking method, the returned message must be checked. If the message is valid, then it will be processed. Otherwise, a *block* method is called. We will see that *block* does not really block the behaviour. This confusing method name could lead programmers to think that block will suspend the behaviour until a new message arrives and the flow will continue from this point. In fact, this is not what happens: The block call only sets a flag in this behaviour, indicating that it will be blocked on further scheduler rounds, until a new message arrives. Additionally, the flow of execution continues normally in the following lines, after the block method execution.

AnswerSender behaviour The behaviour responsible to send the response to the user is very simple. It retrieves the result from the data store, as well as the original message from the user in order to generate the reply. A new reply message is created and sent to the end-user.

OMAS

The OMAS agent does the same thing as the JADE agent. The source-code shown Figure 14.7 presents the agent configuration and skills definition. The difference with the MULTIPLY skill is now that the :dumb-fac skill has two parts (a static part (Figure 14.8) and a dynamic part (Figure 14.9). The static part is executed the first time around and the dynamic part is executed whenever the agent gets an answer to a subtask.

The static part of the skill sends a subtask to one of the MULTIPLY agent chosen randomly. It then keeps the updated value of nn in its environment (a user-formatted area to be used while executing a skill).

```
1  // Package and import declarations removed for the sake of clarity
2
3  public class FacAgentV0 extends Agent {
4
5     protected void setup() {
6        // adding a behaviour to attend user requests
7        addBehaviour(new RequestPerformer());
8     }
9
10    /**
11     * Waits for end-user's messages and
12     * route the flow to more specializes
13     * behaviours.
14     */
15    private class RequestPerformer extends CyclicBehaviour {
16
17       public RequestPerformer() {
18          // Creating and setting the common datastore
19          setDataStore(new DataStore());
20       }
21
22       public void action() {
23          // filtering and receiving only request messages
24          MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
25          ACLMessage req = receive(mt);
26          if (req != null) {
27
28             // stores the request for further replies
29             getDataStore().put("request", req);
30
31             // get number to calculate
32             int nn = Integer.parseInt(req.getContent());
33
34             // initializing the answer variable
35             getDataStore().put("answer", nn);
36
37             // return automatically the response
38             if(nn <= 2) {
39                addBehaviour(new AnswerSender(getDataStore()));
40             } else {
41                // send message to multiplier
42                getDataStore().put("nn", nn);
43                addBehaviour(new MultiplySender(getDataStore()));
44             }
45          } else {
46             block();
47          }
48       } // end of action
49    } // end of inner class
50 } // end of class
```

Figure 14.3: Java code for the FACTORIAL agent class

When the final result is obtained, the skill exits through the API dynamic-exit function, which returns the result to whoever asked for it.

Comparing JADE and OMAS

The same remarks that were made for the MULTIPLY agent apply here. We can add the following.

```
1
2     /**
3      * Used to send messages to multipliers.
4      *
5      */
6     public class MultiplySender extends OneShotBehaviour {
7
8        public MultiplySender(DataStore dataStore) {
9           setDataStore(dataStore);
10       }
11
12       public void action() {
13          int answer = (Integer) getDataStore().get("answer");
14          int nn = (Integer) getDataStore().get("nn");
15
16          ACLMessage mulReq = new ACLMessage(ACLMessage.REQUEST);
17          int agentNb = new Random().nextInt(1) + 1;
18          mulReq.addReceiver(new AID("MUL" + agentNb, AID.ISLOCALNAME));
19          mulReq.setContent(answer + ";" + (nn - 1));
20          mulReq.setReplyWith(Long.toString(System.currentTimeMillis()));
21          send(mulReq);
22
23          addBehaviour(new MultiplyReceiver(getDataStore()));
24       }
25    }
```

Figure 14.4: Multiply sender behaviour for the factorial agent

- Program values

  In JADE, when dealing with more complicated problems involving the exchange between behaviours, is it possible to use the *data store* feature, as shown in the example. OMAS has a similar functionality, allowing the usage of a transient or glue area for the exchange of information between skill functions. However each platform operates in different scopes: The OMAS transient area is related to the underlying thread, meaning that information exchange is allowed only for the same request. In contrast, the JADE datastore feature is associated with the behaviour object, which could attend for several request. Besides the heap economy in the JADE approach, this feature obliges the programmer to control the key names, giving unique identifiers (such as conversation ids) to prevent isolation problems.

- Processes

  By default, a JADE agent has a single thread of execution, which could represent a bottleneck even in multicore environments. Additionally, the architecture suggests the creation of fine-grained behaviours, with low-level of complexity and dependency. Much of the effort is transferred to the programmer. Inexperienced programmers could easily create defective agents that sometimes could not deal with multiple behaviours, or present a high CPU consumption due to the usage of polling.

  The OMAS agent has a thread for executing the specific skill whenever the skill is requested. Thus, an OMAS agent can process several messages requiring the same skill, in parallel. For each message a new thread is created. The dynamic-exit function cleans up the corresponding process. A timer attached to each process kills the process after one hour (default), in case the process is hanging.

```
1
2     /**
3      * Waits for a multiplier response
4      */
5     public class MultiplyReceiver extends SimpleBehaviour {
6
7        private boolean done;
8
9        public MultiplyReceiver(DataStore dataStore) {
10          setDataStore(dataStore);
11       }
12
13       public void action() {
14
15          MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
16          ACLMessage mulResp = receive(mt);
17
18          if (mulResp != null) {
19             int answer = Integer.parseInt(mulResp.getContent());
20             getDataStore().put("answer",answer);
21
22             int nn = (Integer) getDataStore().get("nn");
23             nn --;
24
25             if(nn == 1) {
26                addBehaviour(new AnswerSender(getDataStore()));
27             } else {
28
29                getDataStore().put("nn", nn);
30                addBehaviour(new MultiplySender(getDataStore()));
31             }
32             done = true;
33
34          } else {
35             block();
36          }
37       }
38
39       public boolean done() {
40          return done;
41       }
42    }
```

Figure 14.5: Multiply receiver behaviour for the factorial agent

14.2.4 Launching the Platform

First one must launch the corresponding platform before loading the agents.

JADE

For example on a UNIX-like environment one first specifies the path where the system can find the JADE library of classes, e.g. by setting a global variable. Here we assume one environment variable JADEDIR, previously configured with the JADE location:

```
export CLASSPATH=$JADEDIR/lib/jade.jar
```

Then one launches the GUI as follows:

```
1   /**
2    * Sends the answer to the user
3    */
4   public class AnswerSender extends OneShotBehaviour {
5
6       public AnswerSender(DataStore dataStore) {
7           setDataStore(dataStore);
8       }
9
10      public void action() {
11          ACLMessage req = (ACLMessage) getDataStore().get("request");
12          int answer = (Integer) getDataStore().get("answer");
13
14          ACLMessage reply = req.createReply();
15          reply.setPerformative(ACLMessage.INFORM);
16          reply.setContent(String.valueOf(answer));
17          send(reply);
18
19          System.out.println("Resp: " + reply.getContent());
20      }
21  }
```

Figure 14.6: Answering behaviour for the factorial agent

```
1  ;;;=============================================================================
2  ;;;04/07/22
3  ;;;                              AGENT FAC
4  ;;;
5  ;;;=============================================================================
6
7  (defpackage :fac (:use :cl :moss :omas))
8  (in-package :fac)
9
10 (omas::defagent FAC :redefine t)
11
12 ;;; ================================ skill section ================================
13
14 (omas::defskill :dumb-fac FAC
15   :static-fcn static-dumb-fac
16   :dynamic-fcn dynamic-dumb-fac
17   )
```

Figure 14.7: OMAS code for the factorial agent - definition

```
java jade.Boot -gui
```

Something like the following lines is printed on the terminal:

```
INFO: -------------------------------------------------------
      This is JADE 4.2.0 - revision 6574 of 2012/06/20 15:38:00
      downloaded in Open Source, under LGPL restrictions,
      at http://jade.tilab.com/
      -------------------------------------------------------
...
INFO: -------------------------------------------------------
      Agent container Main-Container@192.168.1.10 is ready.
      -------------------------------------------------------
```

```
(defun static-dumb-fac (agent message nn)
  ;; if nn is less than or equal to 1, then return 1 immediately
  (if (< nn 2) (static-exit agent 1)
      ;; otherwise, create a subtask for computing the product of the first
      ;; two top values
      (progn
        ;; ship subtask to agent MUL-1 to compute
        (send-subtask agent :to (nth (random 2) '(:MUL-1 :MUL-2))
                      :action :multiply :args (list nn (1- nn)))
        ;; define a tag (:n) in the environment to record the value of the next
        ;; products to compute. e.g., nn-2 -> (nn - 2)!
        (set-env (- nn 2) :n)
        ;; quit
        (static-exit agent :done))))
```

Figure 14.8: OMAS code for the factorial agent - static function

```
(defun dynamic-dumb-fac (agent message answer)
  ;; this function is called whenever we get a result from a subtask. This
  ;; approach is not particularly clever, since the computation is linear
  ;; and uses the same multiplying agent, i.e., MUL-1."
  (let ((nn (env-get agent :n)))
    ;; if the recorded value is 1 or less, then we are through
    (if (< nn 2)
      ;; thus we do a final exit
      (dynamic-exit agent answer)
      ;; otherwise we multiply the answer with the next high number
      ;; creating a subtask
      (progn
        (send-subtask agent :to (nth (random 2) '(:MUL-1 :MUL-2))
                      :action :multiply :args (list answer nn))
        ;; update environment
        (set-env (1- nn) :n)
        ;; then return an answer to nobody in particular
        answer
        ))))
```

Figure 14.9: OMAS code for the factorial agent - dynamic function

... and the Jade GUI appears (Figure 14.10).

OMAS

The OMAS platform is launched by starting the Lisp environment and loading the *load-omas-moss.fasl* for Windows.

Some lengthy text appears in the Lisp listener or CommonGraphics window and the initial window pops up on the screen (Fig. 14.11).

IP address and port number are the local broadcast address and the port used by the platform.

14.2.5 Loading New Agents

JADE

Jade agents are created one by one using the create agent button of the interface (Fig. 14.12). They can also be loaded using the *-agents* startup option.

Fig. 14.13 shows the state of the container when two Multiply agents have been created.

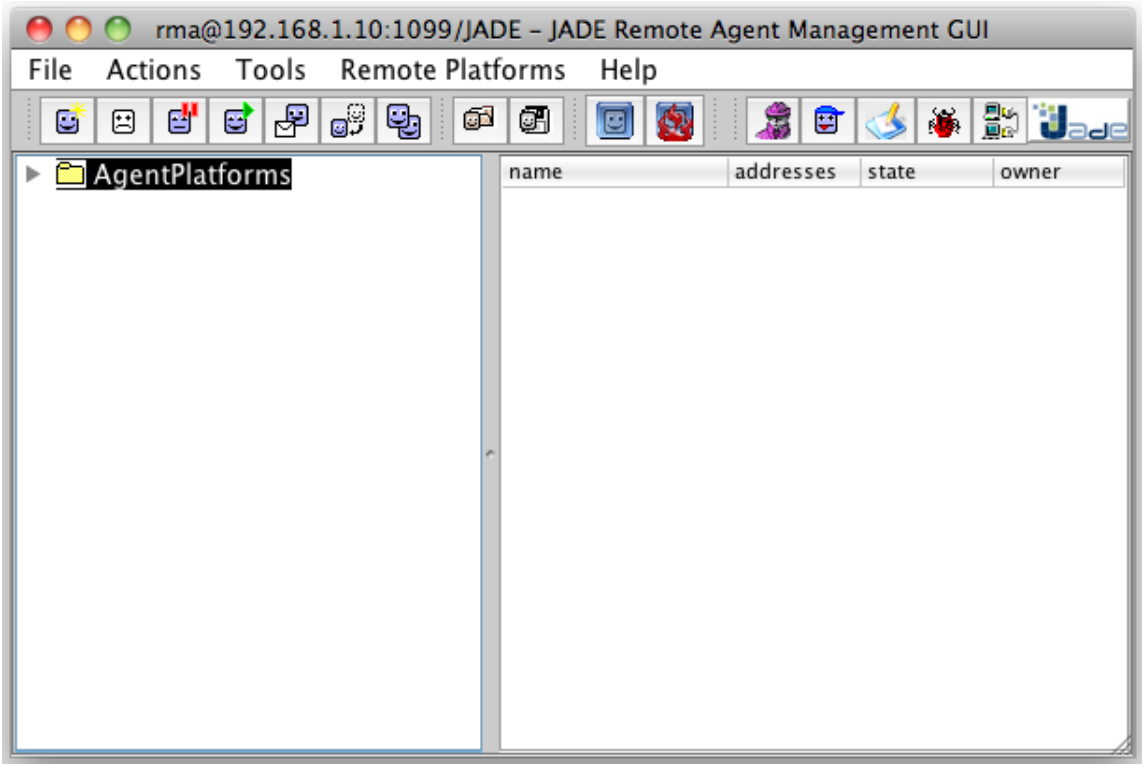

Figure 14.10: The JADE GUI

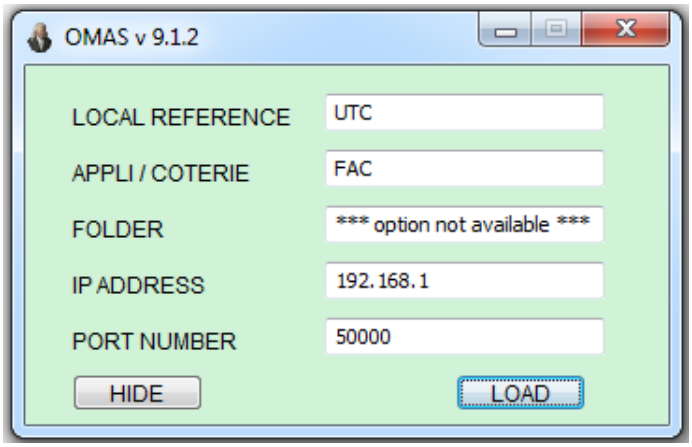

Figure 14.11: The OMAS Initial Window

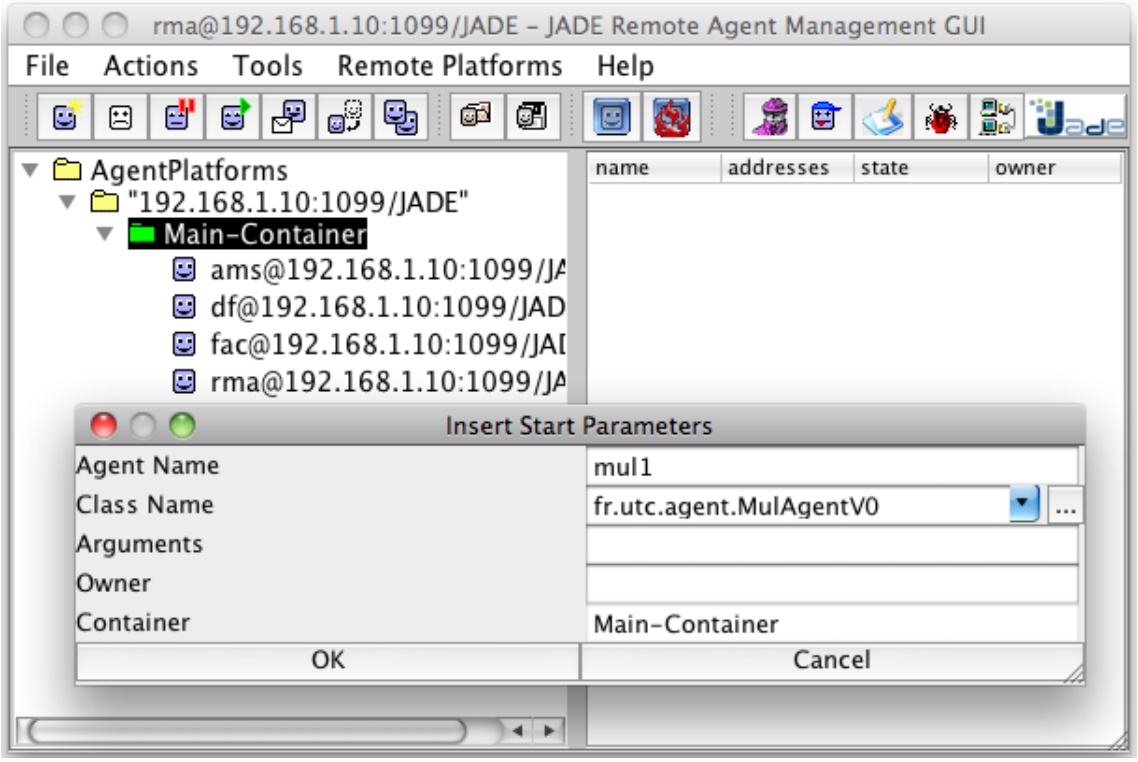

Figure 14.12: Creating a new MULTIPLY agent named mul1

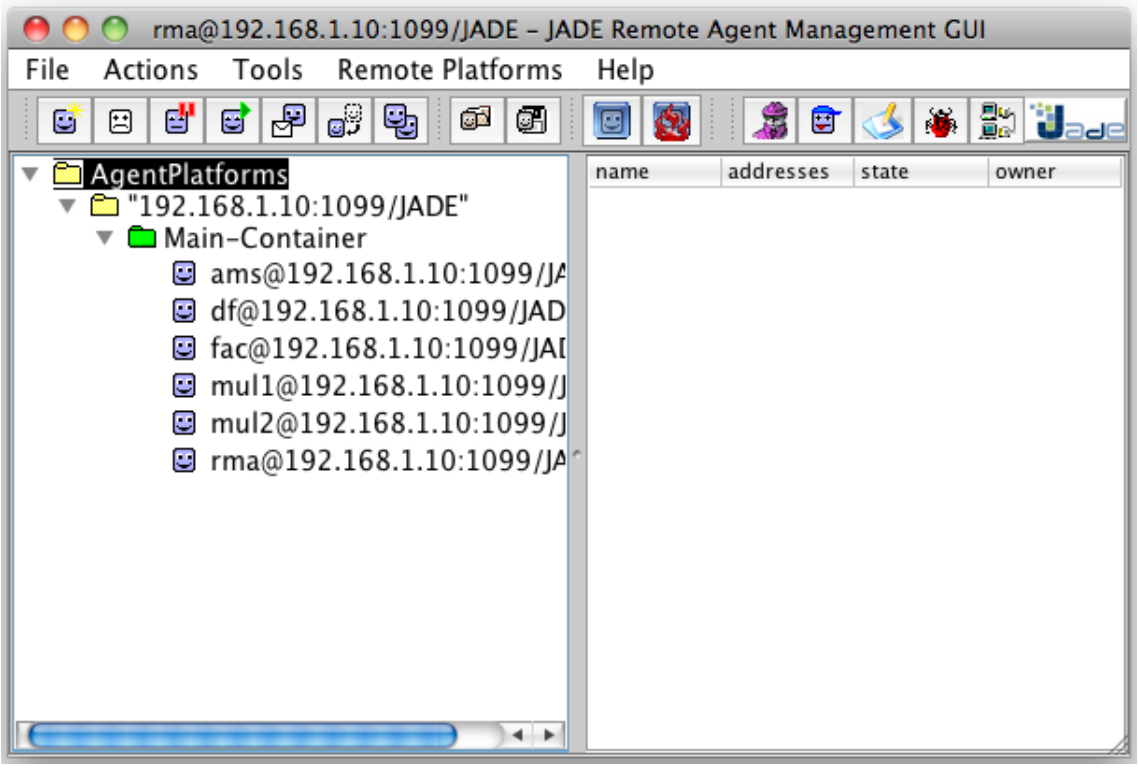

Figure 14.13: Main container with two multiply agents

OMAS

The OMAS agents are loaded by specifying the name of the application file, here, FAC. As soon as the agents are loaded, a control panel appears in the top left corner of the screen (Fig. 14.14).

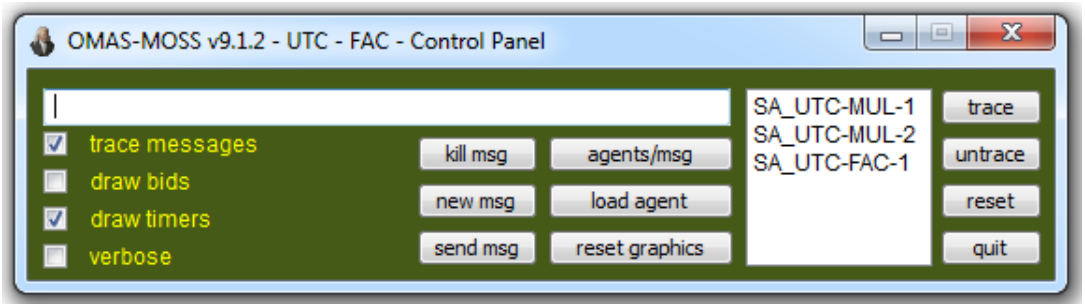

Figure 14.14: OMAS Control Panel for the FAC application

Comparing JADE and OMAS

The main difference is that JADE agents are created as instances of a particular class, and the OMAS agents are loaded as individual entities.

14.2.6 Executing Agents

JADE

In the Jade environment, as soon as an agent is loaded, it executed its behaviours. Thus, when we load the Multiply agents, they execute a cyclic behaviour, waiting for a message containing a multiplication to do. When we load the Factorial agent, then it executes the behaviour for which it is programmed, waits for user requests, gets the results from the multipliers agents and replies with the message containing the factorial result.

OMAS

OMAS agents become active as soon as they are loaded. They do not do anything, waiting for a message. Thus, the Factorial agent will not do anything until it gets a request asking it to compute a factorial. Such a message may be composed with the "new message" button, or preloaded in the *z-messages.lisp* application file, using the defmessage macro.

```
(defmessage :DF-4 :from :<user> :to :FAC :type :request :action :dumb-fac :args (4))
```

As soon as the message is received, the factorial is computed. While computing, the Factorial agent stays active, waiting for another message or something else to do. In other words, the Factorial agent can compute any number of factorials in parallel.

14.2.7 Debugging Agents

Sending Messages Manually

In both platforms it is possible to send messages manually. The Jade GUI has a message button for sending messages. The resulting message composition window can be seen Fig. 14.16 to be compared with that of Fig. 14.15 for OMAS

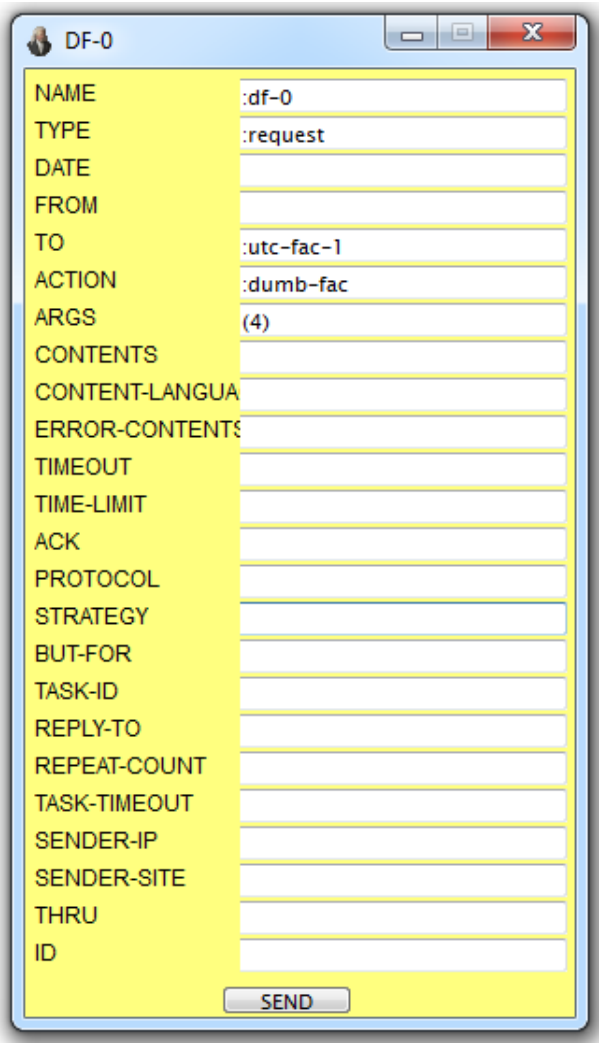

Figure 14.15: Message composition window

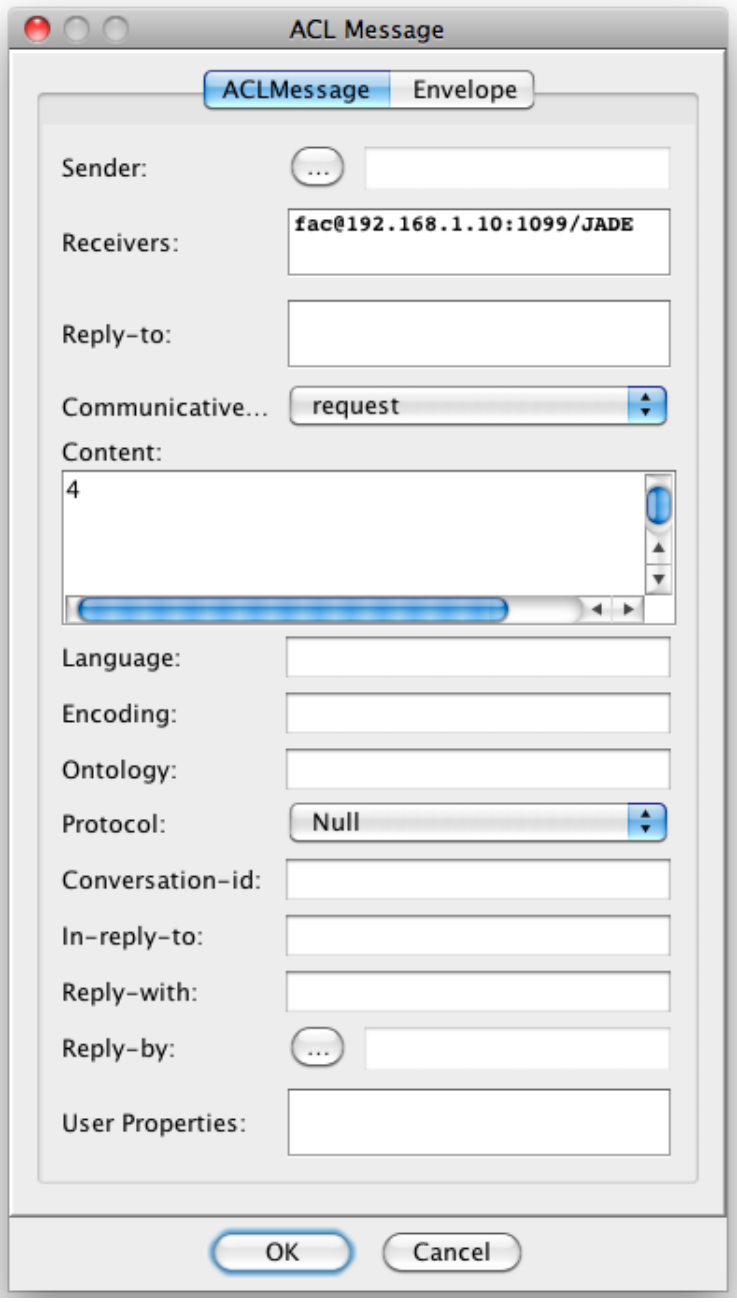

Figure 14.16: Message composition window

Comparing JADE and OMAS

It can be seen that JADE messages are more complex than OMAS messages, but do not contain any field for specifying timeouts or time-limits.

JADE users can also specify an envelope through a second page (Fig. 14.17).

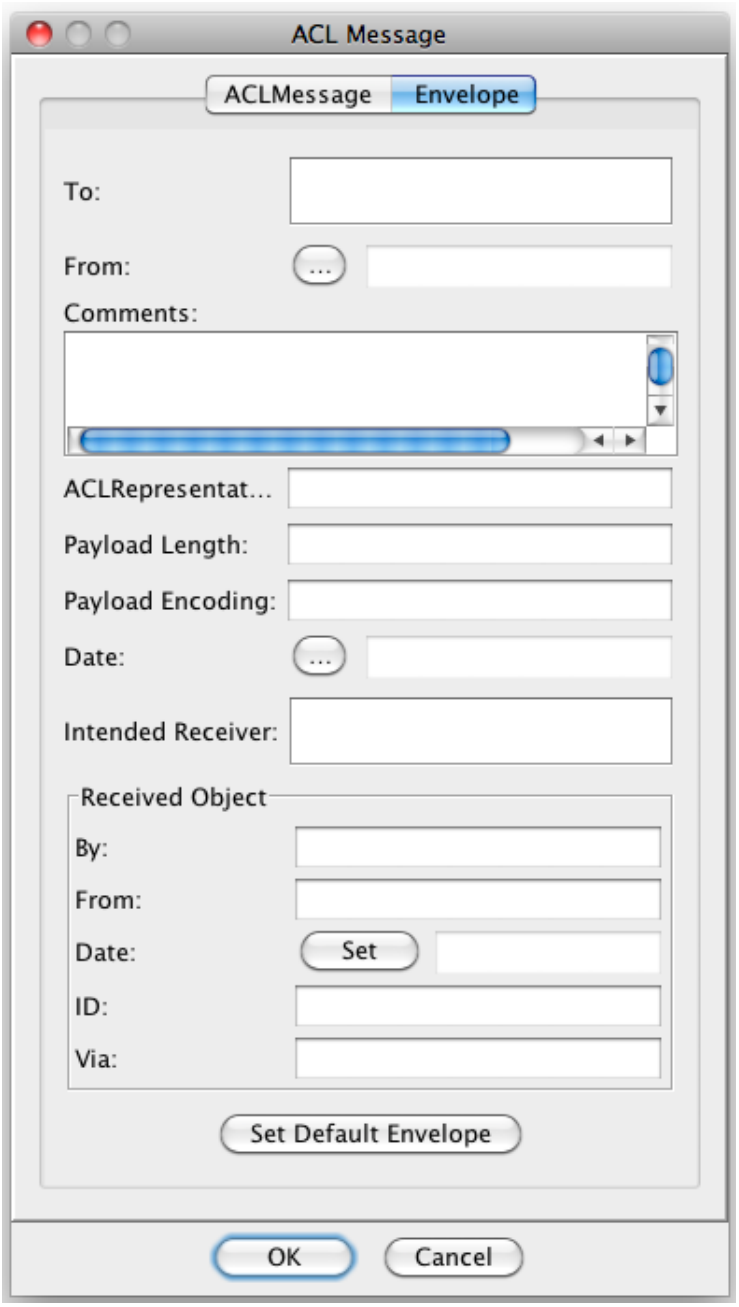

Figure 14.17: JADE message envelope specification window

Tracing Messages

Both platforms have elaborate mechanisms for tracing messages sent to agents. JADE uses a Sniffer agent, OMAS uses the Control Panel to do so.

14.3 Handling Messages

The next point to be studied is how the two platforms handle messages, i.e. how an agent recovers and processes a message.

14.3.1 Receiving Messages

JADE

Recovering messages for a JADE agent is accomplished very simply by invoking the myAgent.receive() function inside a behaviour. This is a non-blocking method, meaning that it will return a null reference if the message queue is empty.

One problem is that each JADE agent has a single thread in order to execute the behaviours. Consequently, a behaviour executes until it relinquishes control willingly, which corresponds to a cooperative rather than preemptive multitasking programming style. Thus, to avoid locks by polling constantly, JADE agents can use the block() function to mark the given behaviour as blocked and let others take control of the execution. As a consequence, blocked behaviours will be skipped by the scheduler, until a new message comes in. The order in which messages are processed must then be specified.

The message queue structure is available to all active behaviours. That is to say, the message originally belongs to the agent and not to the behaviour. As a consequence, for agents with more than one behaviour that uses the receive method, one must explicitly create a message selection mechanism to avoid an error-prone behaviour. One explicit but not effective way of filtering messages is to read the message, evaluate its content and reinsert undesired messages into the queue. The putPack method reinserts a message in front of the queue for further processing. However, the *template* feature is a more effective way of filtering Jade messages.

JADE Message Patterns In order for a specific behaviour to extract the proper messages, the JADE programmer can use a system of patterns called *templates*.

```
1 private class MultiplyServer extends CyclicBehaviour {
2
3    public void action() {
4
5       MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
6
7       ACLMessage msg = myAgent.receive(mt);
```

The above code (from the CyclicBehaviour of the Multiply agents) defines a template to retrieve only REQUEST messages, ignoring all others.

```
1          ...
2
3     // Prepare the template to get results
4
5     mt = MessageTemplate.and(MessageTemplate.MatchConversationId("factorial"),
6
7                              MessageTemplate.MatchInReplyTo(request.getReplyWith()));
```

The above code is extracted from the Factorial agent behavior. The program constructs a template for retrieving just the answers to the message being sent, using a *global conversation id* and the particular tag of the message being sent [request.getReplyWith()].

Using templates a JADE programmer can retrieve the exact messages or answers needed for a correct execution of the corresponding behaviour.

OMAS

OMAS agents work differently. Whenever a message comes in, a specific thread (scan process) scans it to detect whether it should be processed immediately, or whether it must be scheduled for processing with the other tasks. Typical messages that are processed immediately are abort, cancel, inform messages. Typical messages that are scheduled for later processing are request, answer, bid messages. Deferred messages are put into an *agenda* and processed by a special thread (main process). When the agent decides to execute a new task, a new thread is created for this task. The number of tasks (and corresponding threads) that an agent can execute is unlimited. Additional threads are created to accommodate various timers associated with a task.

An OMAS agent can execute several tasks in parallel, as many as its local computer can accommodate.

Regarding message extraction or selection, the incoming messages are *automatically* routed to the proper task (skill function) and task thread. It is not necessary to specify templates as in the JADE context.

A specific feature of the OMAS platform is that all agents receive all messages. Thus, an agent has the possibility to process messages for which it is not a receiver. This feature is a special feature of the *coterie* approach and is useful for knowledge management applications.

14.3.2 Recovering the Message Content (simple case)

JADE

In the simplest case, a JADE agent can recover a message by using the getContent() method on the received message. It then has a simple string of data to be processed within the behaviour. However, a JADE agent can use more elaborate mechanisms to handle messages as will be shown later.

OMAS

The content of a message is transferred to an OMAS agent as additional arguments of the skill functions. The transformation between string format and Lisp s-expression is done automatically. Thus, the processing appears more like a method call as in an OO language. However, this can be more complex.

14.4 Discovering Services

Agents are entities providing services implemented by behaviours or skills. An agent may also require services from other agents. When requiring an unknown service, there are two possible options: either consult yellow pages to find what are the agents providing the service, which is the JADE approach, or sending a broadcast message, which is the OMAS approach.

14.4.1 Service Registration (Yellow Pages)

Registering services is provided by the JADE platform, following the FIPA standards. Thus, an agent usually registers the services it can offer to the platform *Directory Facilitator*.

For example, a Multiply agent registers its services during the setup phase as follows.

```
1 protected void setup() {
2
3     // Register the multiplying service in the yellow pages
4
5     DFAgentDescription dfd = new DFAgentDescription();
6     dfd.setName(getAID());
```

```
7     ServiceDescription sd = new ServiceDescription();
8     sd.setType("multiplying");
9     sd.setName("JADE-multiplying");
10    dfd.addServices(sd);
11
12    try {
13       DFService.register(this, dfd);
14    } catch (FIPAException fe) {
15       throw new RuntimeException(fe);
16    }
17    ...
```

The type of the service is indicated by the "multiplying" string (line 8), the name of the service is "JADE-multiplying" (line 9). Any agent from the platform or from another platform can then use the service, and in particular the Factorial agent by asking the DF platform service.

```
1     // Get the list of multiplying agents
2     DFAgentDescription template = new DFAgentDescription();
3     ServiceDescription sd = new ServiceDescription();
4     sd.setType("multiplying");
5     template.addServices(sd);
6
7     try {
8
9        //DFAgentDescription[] result = DFService.search(myAgent, template);
10       // cannot use myAgent if not inside a behaviour ??
11
12       DFAgentDescription[] result = DFService.search(this, template);
13       System.out.println("Found the following multiplying agents:");
14       mulAgents = new AID[result.length];
15       for (int i = 0; i < result.length; ++i) {
16          mulAgents[i] = result[i].getName();
17          System.out.println("   "+mulAgents[i].getName());
18       }
19    }  catch (FIPAException fe) {
20       throw new RuntimeException(fe);
21    }
22          ...
```

14.4.2 Broadcast

Broadcast is another way of requesting services by sending a message to all agents. Then only the agents capable of providing the service will (eventually) answer.

JADE

JADE has no particular mechanism for broadcasting a message, other than sending the message to all agents after obtaining the names of the agents from the DF service. However, this is not really adapted to the JADE approach. In the JADE approach the programmer will typically select a few agents capable of providing the service and send a message to each agent in turn, waiting for its answer.

OMAS

In the case of OMAS, it is very simple to broadcast a message to all agents of the platform as follows.

```
(send-subtask agent :to :ALL :action :multiply :args (list nn (decf nn)))
```

The agent will then wait some time for possible answers and process them using three possible strategies :take-first-answer, :take-first-n-answers or :collect-answers. In the first case (default strategy) the agent processes the first answer that comes back, in the second and third case answers are collected until a timeout occurs or until we have the number of specified answers. The answers are then analyzed according to the agent strategy. If no message is collected during the allowed time, a timeout error occurs. And by default the waiting task is aborted.

It could be objected that sending a broadcast message is expensive. However, OMAS uses UDP, which means that broadcasting requires a single message.

The broadcast mechanism implies that OMAS agents are gathered locally on the same physical loop or on adjacent loops. The OMAS platform has thus physical boundaries rather than logical boundaries. JADE agents can be located anywhere in the world. To connect distant loops OMAS uses a special transfer agent that may be viewed as a gateway.

14.5 Content Language

The content language is the language specifying the action to be undertaken or the information to be transferred. JADE agents use several languages for exchanging information. OMAS agents use Lisp or natural language.

One of the problems concerns the impedance mismatch between the format of the content of the message and the way it can be processed by the agent. JADE agents live in a Java environment. OMAS agents live in a Lisp environment.

JADE

JADE agents live in a Java environment. Thus, everything must be expressed in terms of objects. However, messages use a string format for network exchange. Consequently, the string format representing the message content must be transformed into Java objects. The Java object must have been predefined in order to be instantiated. Hence, a JADE agent must know a priori all the classes of objects that can be mentioned in a message. This is done by defining an *ontology*.

However, the problem is even worse in the sense that all information about the name of the classes and the names of the properties of the classes is lost after compilation. Thus, it is necessary to build special structures that will contain the names at run time. In other words, it is necessary to define classes as instances of meta-classes, and to keep the names somewhere[4].

If a class is unknown, then it is not possible to build an object that would have been an instance of that class.

Finally, the message content obeys a syntax corresponding to the content language syntax, namely, SL or LEAP. Transforming the content into objects requires modeling the possible performative structures as a set of objects. In JADE this is done by specifying a piece of code known as a *codex*.

For example, the information that there is a person whose name is Giovanni and who is 33 years old in a ACL content expression can be represented as :

```
(Person :name Giovanni :age 33)
```

JADE needs to transform the input string into an instance of a person class:

```
1 class Person {
2    String name;
3    int age;
4
5    public String getName() {return name;}
```

[4]In fact this is not really necessary and JADE agents could use introspection to recover the names of the existing classes. This does not seem to have been used by the JADE developers.

```
6     public void setName(String n) {name = n;}
7     public int getAge() {return age;}
8     public void setAge(int a) {age = a;}
9     ...
10 }
```

The instance should be initialized with:

```
1 name = "Giovanni";
2 age = 33;
```

Thus, each time a JADE agent receives some piece of information, it needs to convert the content of the message into Java objects. At the same time it can verify that the values associated with properties are of the correct type. The conversion is done by the "JADE support for content languages and ontologies" package.

OMAS

OMAS agents accept all contents. Conversion from the string to the list is done by the read-from-string Lisp primitive. OMAS agents do not require a class/instance format. Thus, the JADE problem does not happen.

Most content languages specify performatives as lists. Thus, the translation into a Lisp structure is trivial and does not require any special mechanism. However, there is the need of a specific interpreter to process the resulting list structure.

For example, if we consider the previous example, the list structure is converted automatically and can be used as such, regardless of the existence of a class person.

14.6 Goals and Skills vs. Behaviours

What agents do must be coded somewhere. OMAS uses *goals* and *skills* and JADE uses *behaviours*.

JADE Behaviours

Citing JADE Tutorial (Section 4):

"... the actual job an agent has to do is typically carried out within "behaviours". A behaviour represents a task that an agent can carry out and is represented as an object of a class that extends *jade.core.behaviors.Behaviour*. [...]

Three types of behaviours are available: (i) "One-Shot behaviours that complete immediately; (ii) "Cyclic" behaviours that never complete and execute the same operations each time they are called; and (iii) "Generic" behaviours that can execute different operations according to a status. The last behaviours include special cases like "Sequential," "Parallel," or "FSM (Finite State Machine)" behaviours.

Each class extending Behaviour must implement the action() method, that actually defined the operations to be performed when the behaviour is in execution and the done() method (returns a boolean value that specifies whether or not a behaviour has completed and has to be removed from the pool of behaviours an agent is carrying out."

Thus, one can see the strong influence of the object approach.

A JADE agent can execute several behaviours. However, since there is a single thread, the programmer has to handle all concurrency problems, blocking and unblocking processes, and answering system events.

A behaviour is usually declared in the setup function although it can be added at any time. E.g., for the factorial agent:

```
    // Put agent initializations here
    protected void setup() {
        addBehaviour(new RequestPerformer());
    }
```

The code for RequestPerformer is shown later.

OMAS Skills and Goals

The OMAS approach is quite different and relies on two concept: *skills* and *goals*.

Skills implement code allowing agents to do something, and may be thought as methods for objects (although the activating mechanism is quite different). Skills allow an agent to be *reactive*, i.e. react to messages requiring a task corresponding to one of their skills. Note however that an agent may ignore a request message altogether.

Goals correspond to preset tasks or tasks that will be triggered according to some conditions. They correspond to the *proactive* behavior of the agent. Goals can be one-shot, or cyclic.

Skills can be added to an agent (and not to an agent class) at any time by using the defskill macro. E.g. for the factorial agent:

```
(omas::defskill :dumb-fac FAC
    :static-fcn static-dumb-fac
    :dynamic-fcn dynamic-dumb-fac
)
```

Of course one must write the functions implementing the skill. Note that for skills requiring to subcontract part of the task, a dynamic function must be specified. This separates the setup part of the task with the part reacting to the reception of partial results. Atomic skills do not have dynamic parts.

Once a skill has been given to an agent, the agent can react to any message requiring this skill. Currently, it will answer a request if it has the corresponding skill, displaying a reactive behavior. However, inside the functions implementing the skill there may be some conditions that will prevent the agent to answer (even a request), which distinguishes the agent behavior from an object behavior.

A goal has a different syntax:

```
(defgoal  BUY-BOOK BOOK-BUYER
  :type              :cyclic
  :period            10            ; seconds
  :goal-enable-fcn   enable-buy-book
  :script            goal-buy-book)
```

In the example the goal is defined as a cyclic goal with a 10s period. Every 10 seconds the goal-enable function (enable-buy-book) is run to see if the conditions are met to take action. If so, the goal-buy-book function is executed.

```
;;; return T to enable the goal, NIL to inhibit it

(defun enable-buy-book (agent script)
  "goal is enable each time there is something in the books-to-buy"
  (declare (ignore script))
  (recall agent :books-to-buy)
  )
```

The enable-buy-book simply checks if the agent has a book to buy in its memory.

```
 (defun goal-buy-book (agent)
  (list (make-instance 'omas::message :type :request
                       :from (omas::key agent) :to (omas::key agent)
                       :action :request-performer
                       :args (list (car (recall agent :books-to-buy)))
```

```
6                       :task-id :BUY-BOOK
7                       )))
```

The goal-buy-book makes the agent send a message to itself. The message will be transferred to its agenda and processed as any other task.

A goal may have additional properties like:

- expiration-date: date at which the goal dies
- expiration-delay: time to wait until we kill the goal
- activation-date: date at which the goal should fire (default is now)
- activation-delay: time to wait before activating the goal
- status: waiting, active, dead,...

The type of goals mentioned this far is called *rigid* goals because they are triggered on hard events like a specific date or after some delay. OMAS has a second type of goals, *flexible* goals, that have a different triggering mechanism. Flexible goals have an energy level. They are triggered when the energy level reaches a threshold value. When they are triggered a change function resets their energy level. This kind of goal is useful to model human feelings like anger.

- activation-level: on a 1-100 scale (default 50)
- activation-threshold: on a 1-100 scale (default 50)
- activation-change-fcn: optional function called at each cycle

14.6.1 Comparing JADE and OMAS

Except for flexible goals not available in JADE, one could say that JADE goals are similar to OMAS goals to implement proactive behaviors. JADE does not make a distinction between reactive and proactive behaviors, leaving this task to the programmer.

Triggering conditions (activation on a specific date or waiting delays) seem to be easier to specify in the OMAS syntax.

14.7 Contract-Net

Contract-Net is a standard moderately complex protocol working as follows:

A specific agent, designated hereafter as the manager (its temporary role), wants to subcontract a task. It sends a call for bids either as a broadcast or as a multicast, and waits for answers. Two strategies can be used: (i) to take the first bid (proposal) that is received and grant the task to the corresponding agent; or (ii) better, to wait for several bids (proposals) and select one or several offers that are considered interesting. When no bids or proposals have been received, then usually the task is canceled. When tasks have been granted, then one waits for the answer as usual.

14.7.1 JADE Contract-Net

JADE implements the contract-net as a standard interaction protocol (JADE Programmer's GUIDE, Section 3.5). JADE defines the concept of conversation distinguishing the Initiator role and the Responder role. Initiator behaviours are usually 1:N.

The ContractNetInitiator/responder protocol is described in Section 3.5.2 of the JADE Programmer's GUIDE.

The initiator agent (manager) sollicits proposals from other agents by sending a CFP (Call For Proposals) message. Responders can reply with a PROPOSE message, or a REFUSE message, or a NOT-UNDERSTOOD message. The initiator agent (manager) evaluates the answers and sends ACCEPT-PROPOSAL messages to the agents that are granted the job. Such agents will return an INFORM message with the result, or a FAILURE message after a while.

When the initiator uses the JADE specific behaviour, timeouts can be specified in the reply-by slot of the messages. By default the reply-by is infinite. All messages after the timeout will remain in the private queue of incoming ACL messages of the initiator agent.

The protocol provide callback methods among which handleAllResponses called after the replies to the CFP, and handleAllResultNotifications called after the replies to the ACCEPT-PROPOSAL have been received.

The answer of a Responder agent to the CFP message is built by the ContractNet ResponderBehaviour class. It has a method called prepareResponse that builds a PROPOSE message. When the ACCEPT_PROPOSAL has been received the prepareResultNotification is called instead, in order to prepare the answer (INFORM/DONE) message.

The Behaviour object implementing the protocol contains a dataStore area that can be used to store messages.

14.7.2 OMAS Contract-Net

The Contract-Net protocol is implemented by OMAS by specifying the protocol parameter in the send-subtask API function. E.g.

```
1       ;; if a multiply agent is timed out, then reallocate to another one
2       ;; by simply doing another broadcast
3       (send-subtask agent :to :ALL :action :multiply
4                     :args (omas::args message)
5                     :protocol :contract-net)
```

If nothing else is done, then the system will automatically broadcast a message to all agents of the platform and take the first returned bid, granting the job to the fastest agent (the default strategy is :take-first-answer). The programmer has nothing else to do.

For a better selection the manager should wait before selecting a "good" offer. The programmer has then to specify several things: set the strategy to :collect-answers or :take-first-n-answers in the send-subtask call, add a callback function to the defskill to select-best-answer-fcn parameter. E.g.

```
1       ;; if a multiply agent is timed out, then reallocate to another one
2       ;; by simply doing another broadcast
3       (send-subtask agent :to :ALL :action :multiply
4                     :args (omas::args message)
5                     :protocol :contract-net
6                     :strategy :collect-answers)
```

For example the BOOK example application of the JADE tutorial could use the following code:

```
1  (defskill :REQUEST-PERFORMER BOOK-BUYER
2    :static-fcn static-REQUEST-PERFORMER
3    :dynamic-fcn dynamic-REQUEST-PERFORMER
4    :select-best-answer-fcn select-best-answer-REQUEST-PERFORMER
5    )
6
7  (defun select-best-answer-request-performer (agent message answer-message-list)
8    ;;keep the cheapest offer
9    (declare (ignore agent environment))
10   ;; order the collected answer messages by increasing prices
11   ;; we do that only when there is more tan one answer
12   (when (cdr answer-message-list)
13     (setq answer-message-list
```

```
14            (sort answer-message-list #'<= :key #'omas::content)))
15    ;; select the message corresponding to the lowest price
16    (omas::contents (car answer-message-list))
```

14.7.3 Comparison JADE/OMAS

Both platforms implement the Contract-Net protocol with the same possibilities. The OMAS approach appears to be much simpler to implement, mainly because the system performs all the bookkeeping automatically.

Another difference is that the OMAS manager does not have to know what are the other agents on the platform that could be interested in the bidding.

14.8 Handling Time

In a complex multi-agent platform one needs to handle time. Time can be delays (i.e. to launch a particular action), timeouts (e.g., time an agent is willing to wait for an answer), or time-limits (e.g., allocated time for an agent to produce an answer). Various time delays are usually implemented by timers as separate threads. When the timer fires, then a handler is waken up to deal with the particular condition. Such situations are better handled in multi-threaded environments like OMAS, less easily in single thread environments like JADE.

14.8.1 JADE Delays

JADE implements delays in different ways.

Scheduling Delays

This is implemented by the WakerBehaviour behaviour. The following example from the JADE Tutorial shows how to program a 10 second delay after the agent has been created:

```
1  public class MyAgent extends Agent {
2     protected void setup() {
3        addBehaviour(new WakerBehaviour(this, 10000);
4           protected void handleElapsedTimeout() {
5              // perform operation, e.g. go buy a book
6              myAgent.addBehavior(new RequestPerformer());
7           }
8        } );
9     }
```

The operation to be performed consists in adding the RequestPerformer behaviour.

If the WakerBehaviour behaviour is replaced by the TickerBehaviour behaviour, then the operation is performed periodically every 10 seconds.

One should be careful about using a waiting function inside a behaviour, since there is a single thread of execution. This in essence will block every other behaviour.

14.8.2 OMAS Delays

OMAS delays could be specified at the goal level, by using the activation-delay parameter, although this could be used on very special occasions. Delays are normally specified by using the delay parameter of the send-subtask API function and more generally are a parameter of any type of message. The result is that the message is sent only after the delay has expired.

Furthermore, within any function, delays are the result of executing the Lisp sleep function, which does not pose any problem since each task executes in its own thread.

Delays are expressed in seconds using integer or floating point arguments (i.e. 0.010 means 10 milliseconds).

14.8.3 JADE Timeouts

A timeout may be set up on the reception of a message by using the class ReceiverBehaviour. Quoting from JADE Programmer's GUIDE (Section 3.4.10):

"Encapsulates an atomic operation which realizes the "receive" action. Its action terminates when a message is received. [...] Two more constructors take a timeout value as argument, expressed in milliseconds; a ReceiverBehaviour created using one of these two constructors will terminate after the timeout has expired, whether a suitable message has been received or not. A Handle object is used to access the received ACL message; when trying to receive the message suitable exceptions can be thrown if no message is available or the timeout expired without any useful reception."

14.8.4 OMAS Timeouts

A timeout delay on a subtask (corresponding to the waiting time for the answer message) is specified as a parameter of the send-subtask API function, e.g.,

```
1   ;; ask about the prices of the book
2   (send-subtask  agent :to :all
3                  :action :offer-request
4                  :args (list title)
5                  :timeout 10    ; allow 10 seconds to answer
6                  )
```

In practice any message can take a timeout parameter. The default behavior after a timeout occurs consists in sending the subtask two more times, then quit, aborting the main task if nothing returns after the last sent message.

The user can however specify a timeout handler for a given skill by using the timeout-handler parameter of the defskill macro, e.g.

```
1  (defskill :fast-fac-with-timeout fac
2    :static-fcn static-fast-fac-with-timeout
3    :dynamic-fcn dynamic-fast-fac-with-timeout
4    :timeout-handler timeout-fast-fac-with-timeout)
```

Of course one needs to write the corresponding function:

```
1  (defun timeout-fast-fac-with-timeout (agent message)
2   ;;function for handling timeout errors, rescheduling multiply subtask as a
3   ;; subtask using a contract-net protocol
4    (case (omas::action message)
5      (:multiply
6       ;; if a multiply agent is timed out, then reallocate to another one
7       ;; by simply doing another broadcast
8       (send-subtask agent :to :ALL :action :multiply
9                     :args (omas::args message)
10                    :protocol :contract-net
11                    :timeout *multiply-timeout*)
12      ;; exit
13      :done)
14    ;; for any other subtask type we let the system process the timeout condition
15     (otherwise
16      :unprocessed)))
```

Here the handler must process all types of subtasks that could be used by the skill, hence the check on the action part of the message.

14.8.5 JADE Time Limits

JADE does not seem to have time-limits in the OMAS sense, i.e. a limit given to a subcontracting agent to do the job. This would have to be implemented by the programmer.

14.8.6 OMAS Time Limits

A *time-limit* is associated with each OMAS message to limit the time a subcontracted task will take to execute. A default value is 1 hour, meaning that if the agent cannot do the job within one hour, then it will abort the task. This is done to avoid orphan tasks in the system, since our agents never die.

The time-limit can be specified through the time-limit parameter of the API send-subtask function.

14.8.7 JADE/OMAS Comparison

OMAS has a richer set of available features concerning time handling situations. This comes probably from the fact that OMAS agents are permanent and have to handle time constraints. JADE agents on the other hand delegates much of the effort to the programmer.

14.9 Executing Several Tasks Concurrently

The execution mechanism is quite different for JADE and for OMAS.

14.9.1 JADE Concurrency

For different reasons including easy migration to another machine, JADE agents execute their behaviours within a single thread. The programmer has to handle all concurrency problems.

A JADE agent executes a set of behaviours in a round-robin fashion non preemptively, meaning that a specific behaviour executes until the end before another one can execute. The programmer however can introduce functions relinquishing control so that another behavior can be started.

An agent is a finite state machine that can be in different states as shown Fig. 14.18.

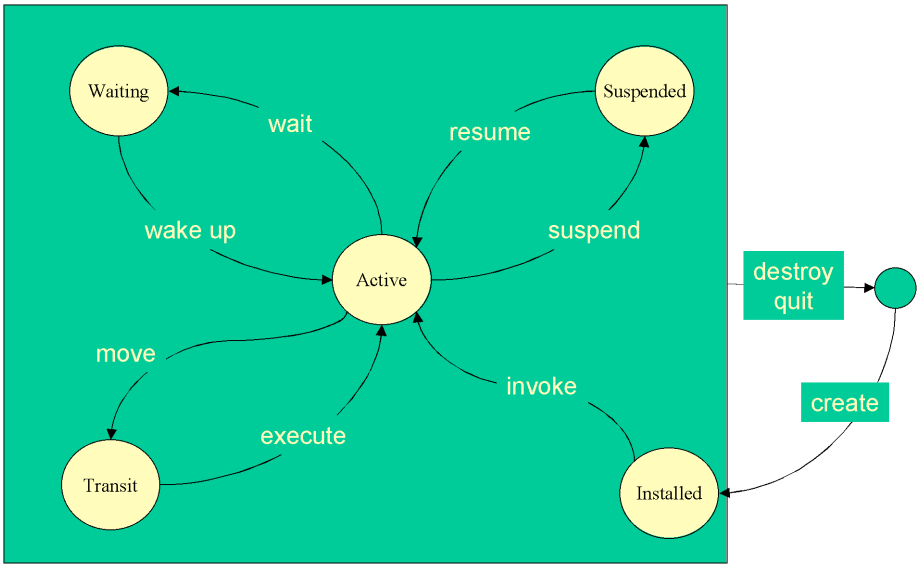

Figure 14.18: Different states of a JADE agent

When an agent is created its setup function installs one or more behaviours. The first one is automatically executed and the agent becomes active. Normally, the behaviour is carried out until it finishes.

Each behaviour waits for an incoming message. Thus, a behaviour could prevent other behaviors from running while in a loop for an incoming message. To avoid such a situation the programmer can use the block() function inside the action() method of a behavior, marking them as blocked. As a result, blocked behaviours are not selected by the scheduler until a new message arrives in the queue. This is done as follows:

```
1  public void action() {
2    ACLMessage msg = myAgent.receive();
3    if (msg != null) {
4      // Message received. Process it.
5      ...
6    }
7    else {
8      block();
9    }
10 }
```

Using this approach, when a message comes in all behaviours are called in turn until one is found that can process the message, in which case its action() method is executed until it returns. Another method can be used to wait for a message, namely blockingReceive(). However, this method blocks the agent thread preventing other behaviours to run until a message is received. The code below shows how simple a behaviour could have its own thread. Thus, to improve the responsiveness level of this kind of behaviour, the usage of the receive() method is preferable. Figure 14.19 illustrates the single-thread model for behaviours execution in JADE. In this example we have an agent with two behaviours with message reading skills, sharing the same thread. Their execution is controlled by the JADE scheduler. The responsiveness level of this model has a high dependence on the programmer approach.

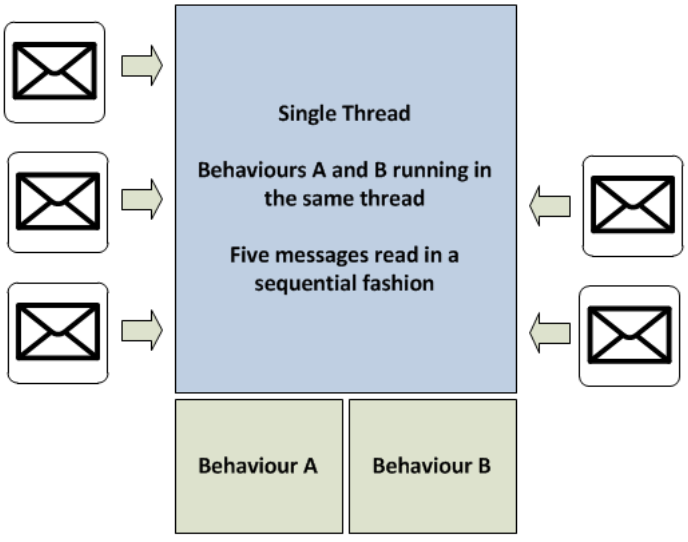

Figure 14.19: JADE single thread model for behaviour execution

One feature available since version 4 of JADE, allows the behaviours to run in a dedicated Java thread. In order to do so, the programmer must wrap a behaviour using the *ThreadedBehaviourFactory* utility class. This feature increases the behaviour responsiveness in multicore environments as well

as allowing the construction of more elegant code when dealing with message reception. Here, the usage of the *blockingReceive* method will not affect the execution of the Jade scheduler, since it runs in a separate thread. However it is up to the programmer to deal with common problems on multithreaded environment such as synchronization and race conditions. The code below shows how simple a behaviour could be having its own thread.

```
1
2 ThreadedBehaviourFactory tbf = new ThreadedBehaviourFactory();
3
4 addBehaviour(tbf.wrap(new MyBehaviour());
```

Using this feature, the scheduling is managed by the Java virtual machine instead of the JADE scheduler. This model is closer to the OMAS approach, allowing the programmer to concentrate on the subject instead of managing operational issues. It results in a more readable and cleaner code. An example of such an effect can be seen in the code below. This snippet is the FAC agent, rewritten using a threaded behaviour approach. Please note the use of the *blockingReceive* and the removal of the *block* method, allowing to put all of the logic in a single behaviour instead of the original four behavours created in Section 14.2.

```
1  public class FacAgentV0 extends Agent {
2
3     protected void setup() {
4        ThreadedBehaviourFactory tbf = new ThreadedBehaviourFactory();
5        // wrapping the behaviour
6        addBehaviour(tbf.wrap(new RequestPerformer()));
7     }
8
9     private class RequestPerformer extends CyclicBehaviour {
10
11       public void action() {
12          MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
13
14          // Will block the dedicated thread
15          ACLMessage req = blockingReceive(mt);
16
17          int nn = Integer.parseInt(req.getContent());
18
19          int answer = nn;
20          while(nn > 2) {
21
22             ACLMessage mulReq = new ACLMessage(ACLMessage.REQUEST);
23             int agentNb = new Random().nextInt(1) + 1;
24             mulReq.addReceiver(new AID("MUL" + agentNb, AID.ISLOCALNAME));
25             mulReq.setContent(answer + ";" + (nn - 1));
26             req.setReplyWith(Long.toString(System.currentTimeMillis()));
27             send(mulReq);
28
29             nn --;
30
31             mt = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
32             ACLMessage mulResp = blockingReceive(mt);
33
34             answer = Integer.parseInt(mulResp.getContent());
35          }
36
37          ACLMessage reply = req.createReply();
38          reply.setPerformative(ACLMessage.INFORM);
39          reply.setContent(String.valueOf(answer));
40          send(reply);
41       }
42    }
```

```
43 }
```

Point out that even with this feature, only one thread is created for the behaviour and not for the request as occurs in OMAS. Thus, behaviours that read the message queue should be prepared to run in a cooperative fashion or choose to serialize their execution. Figure 14.20 illustrates this optional model for behaviour execution. In this example we have an agent with two behaviours, each one having its own thread, controlled by the Java VM scheduler. Both behaviours are able to read request messages from external agents and one execution does not affect the other. However, the responsiveness level of the behaviour itself is highly dependent on the programmer's approach.

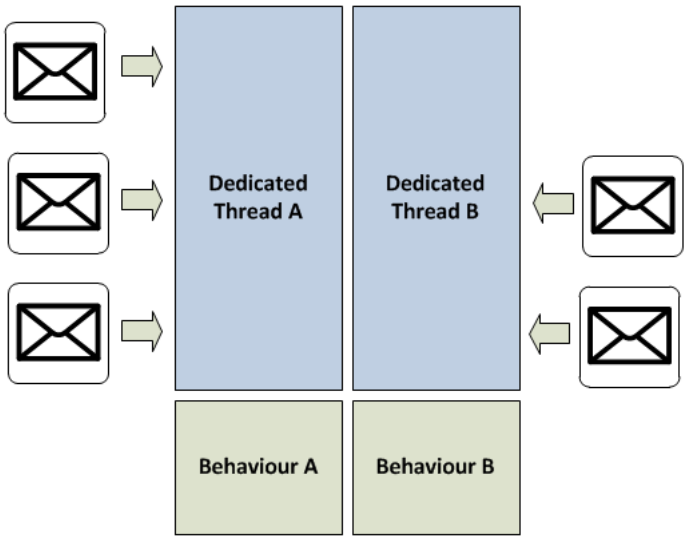

Figure 14.20: Threaded behaviour scenario

14.9.2 OMAS Concurrency

OMAS concurrency is built in since each task executes in its own thread. When an agent receives a message, if the message is urgent or is a quick message (e.g. abort, inform, error, call for bids), then it is executed immediately by a new thread created by the scan process[5]. All other messages are copied into an agenda and are processed by the agent main process. Each request message triggers two processes: (i) a process to execute the corresponding skill; and (ii) a timer process to limit the duration of the corresponding task (by default 1 hour). Thus, if a task needs to wait, it does so in its own process and does not block any other task being active at the same time. Currently there is no limit on the number of processes an agent can have, that is on the number of tasks it can execute in parallel.

Figure 14.21 illustrates OMAS skills execution model. In this example we have an agent with two skills, each one starts a new thread as soon as a message arrives.

Globally, one can consider that an agent has three possible states: (i) idle, waiting for something to do; (ii) active, doing something; or (iii) dead.

[5]Creating a new thread ensures that if the code executing the skill crashes for some reason, it does not take down the scan process.

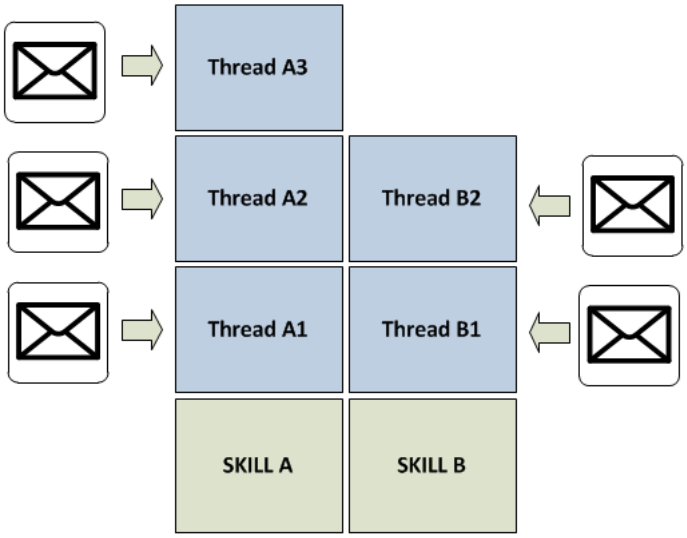

Figure 14.21: OMAS skill execution model

14.10 Ontologies

Ontologies are used to define domain concepts and vocabulary. Domain concepts are used to represent situations or to construct messages. Thus, in order to understand the content of a message, an agent must have the ontology that contain the concept definitions corresponding to the vocabulary used in the message.

14.10.1 JADE Ontologies

A JADE ontology must contain a definition of all the objects and properties that can be handled by the agents. However, since JADE agents deal with Java classes but exchange data as strings, the system will use the ontology to restructure the content of a message as a set of objects. This of course also requires to know which language was used to do the transfer. Thus, JADE ontologies have a complex role and are not disconnected from the programming problems. The result is an intricate machinery that requires some space to be explained and understood.

14.10.2 OMAS Ontologies

OMAS ontologies are defined as a set of frames using the MOSS frame representation language. Marshalling and demarshalling (i.e. transformation between structured representation and linear strings and back) is done by the corresponding Lisp basic functions.

14.11 Problem 0 using WADE

This section presents an alternative implementation of Problem 0, using a subset of JADE. WADE is a library that enables the development of agents using a workflow metaphor. This section is not intended to be a tutorial, but presents a discussion of the resulting WADE-based implementation of Problem 0. A tutorial describing how to install the IDE and build a workflow application can be downloaded from the WADE website (http://jade.tilab.com/wade).

The WADE subset provides a small and lightweight workflow engine based on the JADE library. Basically, the key component of the WADE platform is the *WorkflowEngineAgent* class that extends the Agent class. By the same token, one workflow is represented as a specialized behaviour. The solution also comes with a development environment for Eclipse, called WOLF. The software is available as a plugin and allows common workflow development tasks, such as visual modeling, wizards to create workflows and monitor the server log. Figure 14.22 shows the platform stack.

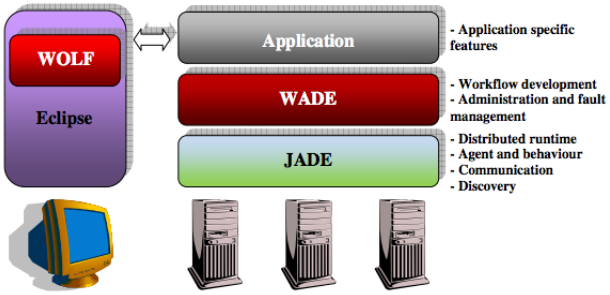

Figure 14.22: The WADE platform stack from the WADE website (http://jade.tilab.com/wade)

Likewise in JADE, our Problem 0 WADE implementation has two classes representing the FAC and MULTIPLY agents. Two behaviours representing the FAC and MULTIPLY workflow were also created. The difference between a normal agent and the workflow one is the level of inheritance: WADE agents must inherit the *WorkflowEngineAgent* and behaviours must inherit the *WorkflowBehaviour*.

Agents Workflow agents have all the known agent features provided by JADE. However, the initialization is slightly different: Workflow Agents are always registered in the DF and workflow information is read from a descriptor file in XML format.

In the workflow paradigm, the agent role is limited, once it starts the workflow (line 10) and retrieves its results (between lines 13 and 42). The framework manages all of the flow control internally. Figure 14.23 shows a code snippet that starts a workflow and retrieves its results using an event-based style (See the interface *WorkflowResultListener and its handleXXX methods*).

Behaviours Following the JADE approach, a behaviour in WADE is used as a central element to implement workflows. The visual modeling tool automatically creates a behaviour for each workflow and maintains its metadata in Java annotations or predefined comments recognized by the tool. Thus, the edition of the source-code is possible but not recommended. Figure 14.24 presents the workflow created for the FAC agent. Each box of the diagram have a java method associated inside the behaviour class. The *WorkflowBehaviour* base-class implements a lightweight workflow engine that controls the sequence of execution and provides utility functions for storing intermediate results, pushing and poping workflow information.

Using the WOLF modeling tool, the user is able to create action and decision nodes. Action nodes can execute simple tasks on the underlying behaviour or execute more sophisticated activities such as start parallel or sequential sub-workflows in other agents or execute predefined web-services. The source-code shown Figure 14.25 is a snippet that starts the calculation workflow on MULTIPLY agents. Please note the usage of the *fill* method, line 3, used to push information inside the workflow.

WADE presents an interesting approach for workflow development in a distributed environment. Citing the WADE tutorial: "This approach makes it possible to combine the expressiveness of the workflow metaphor with the power of a programming language such as Java, and enables the usage of workflows for system internal logics definition." The WYSIWYG approach adopted by WADE

```
1   /**
2    * The method invoked when the user requests
3    * the factorial calculus
4    */
5   void calculate(final long number) {
6      // Prepare the WorkflowDescriptor including the workflow class
7      // and INPUT parameters
8      Map<String,Object> params = new HashMap<String,Object>();
9      params.put("number", number);
10     WorkflowDescriptor wd = new WorkflowDescriptor("fr.utc.agent.workflow.FactorialWorkflow", pa
11     try {
12        // Dispatch the workflow to myself
13        dc.launchWorkflow(getAID(), wd, new WorkflowResultListener() {
14           public void handleAssignedId(AID executor, String executionId) {
15              // The workflow was properly loaded and a unique ID was assigned to it
16              ...
17           }
18
19           public void handleLoadError(String reason) {
20              // The workflow could not be loaded
21              ...
22           }
23
24           public void handleNotificationError(AID executor, String executionId) {
25              // There was a communication error receiving the notification from the executor
26              ...
27           }
28
29           public void handleExecutionError(ExecutionError er, AID executor, String executionId)
30              // The execution of the workflow failed
31              ...
32           }
33
34           public void handleExecutionCompleted(jade.util.leap.List results, AID executor, String
35              // The workflow was successfully executed
36              Map<String,Object> params = ElementDescriptor.paramListToMap(results);
37              Long result = (Long) params.get("result");
38
39              // Send the result to the user
40              ...
41           }
42        }, null);
43     } catch (WorkflowException e) {
44        throw new RuntimeException(e);
45     }
46  }
```

Figure 14.23: Starting a workflow and waiting for the results

facilitates the modeling and building of workflows based on predictable multi-agent environments, but auto-generated code lacks efficiency, and does not provide good readability of the source-code.

14.12 Implementation Complexity

This section presents a brief discussion of the complexity of the JADE and OMAS implementations of Problem 0. Several techniques could be used to evaluate source-code complexity like cyclomatic complexity, inheritance depth and class coupling for OO languages, just to name a few. Such metrics

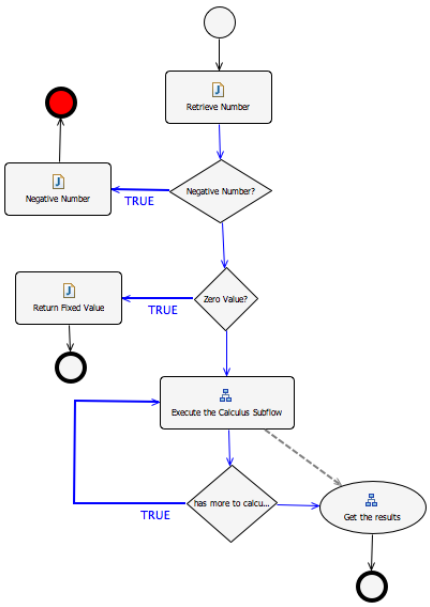

Figure 14.24: The FAC workflow

```
1    protected void executeExecuteCalculusSubFlow(Subflow s) throws Exception {
2
3        s.fill("number", number);
4        s.setPerformer(((ManagerAgent) myAgent).getCalculatorAgent().getLocalName());
5
6        performSubflow(s);
7
8        number -= 2;
9    }
```

Figure 14.25: The source-code of an action that starts a sub-workflow

generally work well when comparing source-codes written in the same language. However, comparing the complexity of two implementations made in different languages is still challenging. Different features affect the comparison, like the language paradigms, level of knowledge of the programmer, cultural aspects and, of course, the syntactical impedance between the languages. For this reason, we choose the SLOC metric (source lines of code) as our lowest common denominator in order to compare the implementations. The comparison is non-conclusive but can give some clues about the programming style of each platform.

Figure 14.26 shows the numbers of lines of source-code for the following implementations: (i) the first JADE implementation using one thread, labeled *JADE (a)* having 136 lines of code; (ii) the second JADE implementation using a thread wrapper, labeled *JADE (b)* having 71 lines of code; (iii) the OMAS implementation having 46 lines of code; (iv) the WADE (a) implementation having 358 lines of code. Finally, the WADE (b) presents the same WADE implementation SLOC count, discounting the source-code generated automatically, resulting in 142 lines of code.

As mentioned above, this metric is non-conclusive but give us some clues, especially when comparing the JADE and OMAS implementations. The first JADE implementation adopts several programming conventions to deal with behaviours in a single-threaded environment. As a result, the SLOC is almost three times higher compared to the OMAS SLOC. In contrast, the wrapping strategy adopted in the second implementation of JADE reduced significantly the amount of operational source-code,

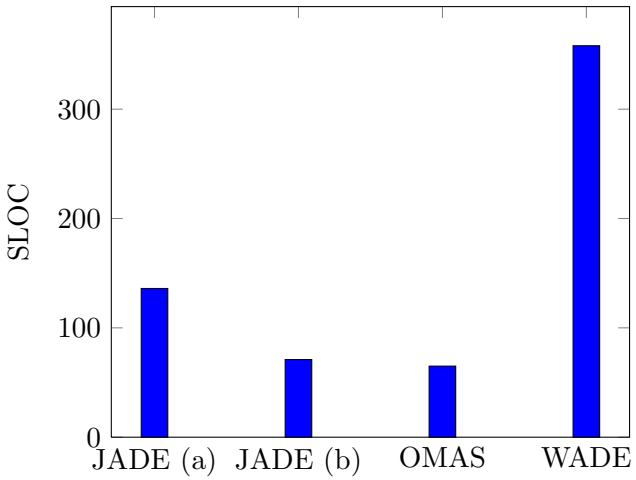

Figure 14.26: Complexity of Implementations using SLOC

specially when dealing with messages, transferring the scheduling issues to the virtual machine. The resulting SLOC was comparable to the OMAS implementation. The WYSIWYG approach adopted by WADE, facilitates the modeling and build of workflow-based agents, but auto-generated code lacks on efficiency or readability. It is also hard to improve as it takes a great effort to understand it. The high SLOC is a common symptom of this programming paradigm. Note that even removing the generated source-code, the SLOC is high, compared to other implementations. One reason could be the event-based programming style when recovering information from workflows (see Figure 14.23) and also the necessary source-code to deal with the directory facilitator agent.

14.13 Appendix

14.13.1 Complete listing of WADE Problem 0 source-code

FactorialAgent.java Source-code of Factorial Agent.

```
1  package fr.utc.agent.manager;
2
3  import jade.core.AID;
4  import jade.domain.DFService;
5  import jade.domain.FIPAException;
6  import jade.domain.FIPAAgentManagement.DFAgentDescription;
7  import jade.domain.FIPAAgentManagement.SearchConstraints;
8  import jade.domain.FIPAAgentManagement.ServiceDescription;
9  import jade.lang.acl.ACLMessage;
10 import jade.proto.SubscriptionInitiator;
11
12 import java.util.ArrayList;
13 import java.util.HashMap;
14 import java.util.List;
15 import java.util.Map;
16
17 import javax.swing.JOptionPane;
18
19 import com.tilab.wade.commons.AgentInitializationException;
20 import com.tilab.wade.dispatcher.DispatchingCapabilities;
21 import com.tilab.wade.dispatcher.WorkflowResultListener;
22 import com.tilab.wade.performer.WorkflowEngineAgent;
23 import com.tilab.wade.performer.WorkflowException;
```

```
24 import com.tilab.wade.performer.descriptors.ElementDescriptor;
25 import com.tilab.wade.performer.descriptors.WorkflowDescriptor;
26 import com.tilab.wade.performer.ontology.ExecutionError;
27
28 public class FactorialAgent extends WorkflowEngineAgent {
29
30    private static final long serialVersionUID = -532078065976693536L;
31
32    private FactorialAgentGui myGui;
33    private DispatchingCapabilities dc = new DispatchingCapabilities();
34    private List<AID> calculatorAgents = new ArrayList<AID>();
35    private int index = 0;
36
37    /**
38     * Agent initialization
39     */
40    protected void agentSpecificSetup() throws AgentInitializationException {
41       super.agentSpecificSetup();
42
43       // Create and show the gui
44       myGui = new FactorialAgentGui(this);
45       myGui.initGui();
46       myGui.setVisible(true);
47
48       // Initialize the DispatchingCapabilities instance used
49       // to launch workflows
50       dc.init(this);
51
52       // Subscribe to the DF to keep the fatorial agent list up to date
53       ServiceDescription sd = new ServiceDescription();
54       sd.setType("Calculator-Agent");
55       DFAgentDescription dfTemplate = new DFAgentDescription();
56       dfTemplate.addServices(sd);
57       SearchConstraints sc = new SearchConstraints();
58       sc.setMaxResults(new Long(-1));
59       ACLMessage subscribe = DFService.
60             createSubscriptionMessage(this, getDefaultDF(), dfTemplate, sc);
61       addBehaviour(new SubscriptionInitiator(this, subscribe) {
62          private static final long serialVersionUID = 2382141579395624376L;
63
64          protected void handleInform(ACLMessage inform) {
65             try {
66                DFAgentDescription[] dfds = DFService.decodeNotification(inform.getContent());
67                for (int i = 0; i < dfds.length; ++i) {
68                   AID aid = dfds[i].getName();
69                   if (dfds[i].getAllServices().hasNext()) {
70                      // Registration/Modification
71                      if (!calculatorAgents.contains(aid)) {
72                         calculatorAgents.add(aid);
73                         System.out.println("Calculator Agent "+
74                            aid.getLocalName()+" added to the list of searcher agents");
75                      }
76                   } else {
77                      // Deregistration
78                      calculatorAgents.remove(aid);
79                      System.out.println("Calculator Agent "+aid.getLocalName()+
80                         " removed from the list of searcher agents");
81                   }
82                }
83             }
84             catch (FIPAException fe) {
```

```
85                  fe.printStackTrace();
86              }
87          }
88      } );
89  }
90
91  /**
92   * Agent clean-up
93   */
94  protected void takeDown() {
95      // Turn off the GUI on agent termination
96      if (myGui != null) {
97          myGui.dispose();
98          myGui.setVisible(false);
99      }
100 }
101
102 public AID getCalculatorAgent() {
103     if (calculatorAgents.isEmpty()) {
104         throw new RuntimeException("No Calculator Agents available");
105     }
106     if (index >=calculatorAgents.size()) {
107         index = 0;
108     }
109     return (AID) calculatorAgents.get(index++);
110 }
111
112 /**
113  * The method invoked by the GUI when the user requests
114  * the factorial calculus
115  */
116 void calculate(final long number) {
117     // Prepare the WorkflowDescriptor including the workflow class
118     // and INPUT parameters
119     Map<String,Object> params = new HashMap<String,Object>();
120     params.put("number", number);
121     WorkflowDescriptor wd = new WorkflowDescriptor("fr.utc.agent.workflow.FactorialWorkflow", pa
122     try {
123         // Dispatch the workflow to myself
124         dc.launchWorkflow(getAID(), wd, new WorkflowResultListener() {
125             public void handleAssignedId(AID executor, String executionId) {
126                 // The workflow was properly loaded and a unique ID was assigned to it
127                 System.out.println("Workflow correctly loaded by performer "+executor.getLocalName
128             }
129
130             public void handleLoadError(String reason) {
131                 // The workflow could not be loaded
132                 System.out.println("Error loading the workflow");
133                 JOptionPane.showMessageDialog(null,"Error loading the workflow. "+reason);
134             }
135
136             public void handleNotificationError(AID executor, String executionId) {
137                 // There was a communication error receiving the notification from the executor
138                 System.out.println("Notification error ("+executionId+")");
139                 JOptionPane.showMessageDialog(null,
140                     "Notification error received from performer "+
141                     executor.getName()+" for workflow "+executionId, "Error", JOptionPane.ERROR_MESS
142             }
143
144             public void handleExecutionError(ExecutionError er, AID executor, String executionId)
145                 // The execution of the workflow failed
```

```
146                 System.out.println("Execution error ("+executionId+")");
147                 JOptionPane.showMessageDialog(null,
148                     "Execution error received from performer "+
149                     executor.getName()+" for workflow "+executionId+
150                     " ["+er.getType()+": "+er.getReason()+"]", "Error", JOptionPane.ERROR_MESSAGE);
151             }
152
153             public void handleExecutionCompleted(jade.util.leap.List results, AID executor, String
154                 // The workflow was successfully executed
155                 System.out.println("Execution OK ("+executionId+")");
156                 Map<String,Object> params = ElementDescriptor.paramListToMap(results);
157                 Long result = (Long) params.get("result");
158                 JOptionPane.showMessageDialog(null," result: " + result);
159             }
160         }, null);
161       } catch (WorkflowException e) {
162         throw new RuntimeException(e);
163       }
164     }
165 }
```

FactorialAgentGui.java Source-code of Factorial Agent Gui.

```
1  package fr.utc.agent.manager;
2
3  import java.awt.Dimension;
4  import java.awt.GridBagConstraints;
5  import java.awt.GridBagLayout;
6  import java.awt.HeadlessException;
7  import java.awt.Insets;
8  import java.awt.event.ActionEvent;
9  import java.awt.event.ActionListener;
10 import java.awt.event.WindowAdapter;
11 import java.awt.event.WindowEvent;
12
13 import javax.swing.JButton;
14 import javax.swing.JFrame;
15 import javax.swing.JLabel;
16 import javax.swing.JPanel;
17 import javax.swing.JTextField;
18 import javax.swing.SwingConstants;
19
20 public class FactorialAgentGui extends JFrame {
21    private static final long serialVersionUID = 1L;
22
23    private FactorialAgent myAssemblerAgent;
24    private JTextField number;
25    private JButton okButton;
26
27    public FactorialAgentGui(FactorialAgent myAssemblerAgent)
28    throws HeadlessException {
29       super("Factorial Agent Gui");
30       this.myAssemblerAgent = myAssemblerAgent;
31       addWindowListener(new    WindowAdapter() {
32          public void windowClosing(WindowEvent e) {
33             FactorialAgentGui.this.myAssemblerAgent.doDelete();
34          }
35       } );
36    }
37
38    public void initGui(){
```

```
39          JPanel rootPanel = new JPanel();
40          rootPanel.setLayout(new GridBagLayout());
41          rootPanel.setMinimumSize(new Dimension(330, 125));
42          rootPanel.setPreferredSize(new Dimension(330, 125));
43
44          JLabel l = new JLabel("Number do calculate:");
45          l.setHorizontalAlignment(SwingConstants.LEFT);
46          GridBagConstraints gridBagConstraints = new GridBagConstraints();
47          gridBagConstraints.gridx = 0;
48          gridBagConstraints.gridy = 0;
49          gridBagConstraints.anchor = java.awt.GridBagConstraints.NORTHWEST;
50          gridBagConstraints.insets = new java.awt.Insets(5, 3, 0, 3);
51          rootPanel.add(l, gridBagConstraints);
52
53          number = new JTextField(10);
54          number.setMinimumSize(new Dimension(222, 20));
55          number.setPreferredSize(new Dimension(222, 20));
56
57          gridBagConstraints = new GridBagConstraints();
58          gridBagConstraints.gridx = 1;
59          gridBagConstraints.gridy = 0;
60          gridBagConstraints.gridwidth = 3;
61          gridBagConstraints.anchor = GridBagConstraints.NORTHWEST;
62          gridBagConstraints.insets = new Insets(5, 3, 0, 3);
63          rootPanel.add(number, gridBagConstraints);
64
65          okButton = new JButton("Ok");
66          okButton.addActionListener(new ActionListener(){
67             public void actionPerformed(ActionEvent e) {
68                Long extractedNumber = Long.valueOf(number.getText());
69                FactorialAgentGui.this.myAssemblerAgent.calculate(extractedNumber);
70
71             }
72          } );
73          gridBagConstraints = new GridBagConstraints();
74          gridBagConstraints.gridx = 0;
75          gridBagConstraints.gridy = 1;
76          gridBagConstraints.anchor = java.awt.GridBagConstraints.NORTHWEST;
77          gridBagConstraints.insets = new java.awt.Insets(5, 3, 0, 3);
78          rootPanel.add(okButton, gridBagConstraints);
79
80          gridBagConstraints = new GridBagConstraints();
81          gridBagConstraints.gridx = 0;
82          gridBagConstraints.gridy = 1;
83          gridBagConstraints.anchor = java.awt.GridBagConstraints.SOUTH;
84          gridBagConstraints.insets = new java.awt.Insets(5, 3, 0, 3);
85
86
87          this.add(rootPanel);
88          pack();
89       }
90
91    }
```

MultiplyAgent.java Source-code of Multiply Agent

```
1  package fr.utc.agent.factorial;
2
3  import com.tilab.wade.commons.AgentInitializationException;
4  import com.tilab.wade.commons.AgentType;
5  import com.tilab.wade.performer.WorkflowEngineAgent;
```

```
6
7  public class MultiplyAgent extends WorkflowEngineAgent {
8     private static final long serialVersionUID = 8434296287509626302L;
9
10    /**
11     * Agent initialization
12     */
13    public void agentSpecificSetup() throws AgentInitializationException {
14       super.agentSpecificSetup();
15       setPoolSize(1);
16    }
17
18    /**
19     * Return the type of this agent. This will be
20     * inserted in the default DF description
21     */
22    public AgentType getType() {
23       AgentType type = new AgentType();
24       type.setDescription("Calculator-Agent");
25       return type;
26    }
27 }
```

FactorialWorkflow.java Source-code of Factorial Workflow.

```
1  package fr.utc.agent.workflow;
2
3  import java.util.List;
4  import com.tilab.wade.performer.SubflowList;
5  import com.tilab.wade.performer.SubflowJoinBehaviour;
6  import com.tilab.wade.performer.SubflowDelegationBehaviour;
7  import com.tilab.wade.performer.Subflow;
8  import com.tilab.wade.performer.layout.MarkerLayout;
9  import com.tilab.wade.performer.layout.TransitionLayout;
10 import com.tilab.wade.performer.Transition;
11 import com.tilab.wade.performer.RouteActivityBehaviour;
12 import com.tilab.wade.performer.CodeExecutionBehaviour;
13 import com.tilab.wade.performer.layout.ActivityLayout;
14 import com.tilab.wade.performer.layout.WorkflowLayout;
15 import com.tilab.wade.performer.FormalParameter;
16 import com.tilab.wade.performer.WorkflowBehaviour;
17
18 import fr.utc.agent.manager.FactorialAgent;
19
20 @WorkflowLayout(exitPoints = { @MarkerLayout(position = "(480,538)", activityName = "FactorialWor
21 public class FactorialWorkflow extends WorkflowBehaviour {
22
23    public static final String
24       FACTORIALWORKFLOWSUBFLOWJOINACTIVITY1_ACTIVITY = "FactorialWorkflowSubflowJoinActivity1";
25
26    public static final String
27       FACTORIALWORKFLOWROUTEACTIVITY3_ACTIVITY = "FactorialWorkflowRouteActivity3";
28
29    public static final String
30       HASMORE_CONDITION = "hasMore";
31
32    public static final String
33       EXECUTECALCULUSSUBFLOW_ACTIVITY = "ExecuteCalculusSubFlow";
34
35    public static final String
36       NEGATIVENUMBER_ACTIVITY = "NegativeNumber";
```

```
37
38      public static final String
39          FACTORIALWORKFLOWROUTEACTIVITY2_ACTIVITY = "FactorialWorkflowRouteActivity2";
40
41      public static final String
42          ISNEGATIVE_CONDITION = "isNegative";
43
44      public static final String
45          RETURNFIXEDVALUE_ACTIVITY = "ReturnFixedValue";
46
47      public static final String
48          FACTORIALWORKFLOWROUTEACTIVITY1_ACTIVITY = "FactorialWorkflowRouteActivity1";
49
50      public static final String
51          ZEROVALUE_CONDITION = "ZeroValue";
52
53      public static final String
54          RETRIEVENUMBER_ACTIVITY = "RetrieveNumber";
55
56      private static final long serialVersionUID = 8587889613885074370L;
57
58      @FormalParameter
59      private Long number;
60
61      @FormalParameter(mode=FormalParameter.OUTPUT)
62      private Long result;
63
64      @SuppressWarnings("unused")
65      private void defineActivities() {
66          CodeExecutionBehaviour retrieveNumberActivity = new CodeExecutionBehaviour(
67                  RETRIEVENUMBER_ACTIVITY, this);
68          registerActivity(retrieveNumberActivity, INITIAL);
69          RouteActivityBehaviour factorialWorkflowRouteActivity1Activity =
70              new RouteActivityBehaviour(
71                  FACTORIALWORKFLOWROUTEACTIVITY1_ACTIVITY, this);
72          registerActivity(factorialWorkflowRouteActivity1Activity);
73          CodeExecutionBehaviour returnFixedValueActivity =
74              new CodeExecutionBehaviour(
75                  RETURNFIXEDVALUE_ACTIVITY, this);
76          registerActivity(returnFixedValueActivity, FINAL);
77          RouteActivityBehaviour factorialWorkflowRouteActivity2Activity =
78              new RouteActivityBehaviour(
79                  FACTORIALWORKFLOWROUTEACTIVITY2_ACTIVITY, this);
80          registerActivity(factorialWorkflowRouteActivity2Activity);
81          CodeExecutionBehaviour negativeNumberActivity =
82              new CodeExecutionBehaviour(
83                  NEGATIVENUMBER_ACTIVITY, this);
84          negativeNumberActivity.setError(true);
85          registerActivity(negativeNumberActivity, FINAL);
86          SubflowDelegationBehaviour executeCalculusSubFlowActivity =
87              new SubflowDelegationBehaviour(
88                  EXECUTECALCULUSSUBFLOW_ACTIVITY, this);
89          executeCalculusSubFlowActivity.setAsynch();
90          executeCalculusSubFlowActivity.setSubflow(MultiplyWorkflow.class
91                  .getName());
92          registerActivity(executeCalculusSubFlowActivity);
93          RouteActivityBehaviour factorialWorkflowRouteActivity3Activity =
94              new RouteActivityBehaviour(
95                  FACTORIALWORKFLOWROUTEACTIVITY3_ACTIVITY, this);
96          registerActivity(factorialWorkflowRouteActivity3Activity);
97          SubflowJoinBehaviour factorialWorkflowSubflowJoinActivity1Activity =
```

```
98              new SubflowJoinBehaviour(
99                  FACTORIALWORKFLOWSUBFLOWJOINACTIVITY1_ACTIVITY, this);
100         factorialWorkflowSubflowJoinActivity1Activity
101                 .addSubflowDelegationActivity(EXECUTECALCULUSSUBFLOW_ACTIVITY,
102                         SubflowJoinBehaviour.ALL);
103         registerActivity(factorialWorkflowSubflowJoinActivity1Activity, FINAL);
104     }
105 
106     /**
107      * Retrieve the number to calculate<br>
108      * @layout visible=false;position=(502,64)
109      */
110     protected void executeRetrieveNumber() throws Exception {
111         System.out.println("Number: " + number);
112     }
113 
114     /**
115      * Check if the number is zero<br>
116      * @layout visible=false;position=(497,251)
117      */
118     protected boolean checkZeroValue() throws Exception {
119         return number == 0;
120     }
121 
122     @SuppressWarnings("unused")
123     private void defineTransitions() {
124         registerTransition(new Transition(ZEROVALUE_CONDITION, this),
125                 FACTORIALWORKFLOWROUTEACTIVITY1_ACTIVITY,
126                 RETURNFIXEDVALUE_ACTIVITY);
127         registerTransition(new Transition(), RETRIEVENUMBER_ACTIVITY,
128                 FACTORIALWORKFLOWROUTEACTIVITY2_ACTIVITY);
129         registerTransition(new Transition(ISNEGATIVE_CONDITION, this),
130                 FACTORIALWORKFLOWROUTEACTIVITY2_ACTIVITY,
131                 NEGATIVENUMBER_ACTIVITY);
132         registerTransition(new Transition(),
133                 FACTORIALWORKFLOWROUTEACTIVITY2_ACTIVITY,
134                 FACTORIALWORKFLOWROUTEACTIVITY1_ACTIVITY);
135         registerTransition(new Transition(),
136                 FACTORIALWORKFLOWROUTEACTIVITY1_ACTIVITY,
137                 EXECUTECALCULUSSUBFLOW_ACTIVITY);
138         registerTransition(new Transition(), EXECUTECALCULUSSUBFLOW_ACTIVITY,
139                 FACTORIALWORKFLOWROUTEACTIVITY3_ACTIVITY);
140         registerTransition(new Transition(HASMORE_CONDITION, this),
141                 FACTORIALWORKFLOWROUTEACTIVITY3_ACTIVITY,
142                 EXECUTECALCULUSSUBFLOW_ACTIVITY);
143         registerTransition(new Transition(),
144                 FACTORIALWORKFLOWROUTEACTIVITY3_ACTIVITY,
145                 FACTORIALWORKFLOWSUBFLOWJOINACTIVITY1_ACTIVITY);
146     }
147 
148     /**
149      * <br>
150      * Return the value 1<br>
151      * @layout visible=false;position=(19,154)
152      */
153     protected void executeReturnFixedValue() throws Exception {
154         result = 1L;
155     }
156 
157     /**
158      * Check wether the number to calculate is negative<br>
```

```
159      * @layout visible=false;position=(223,206)
160      */
161     protected boolean checkisNegative() throws Exception {
162        return number < 0;
163     }
164
165     /**
166      * Negative number activity.
167      */
168     protected void executeNegativeNumber() throws Exception {
169        throw new Exception("Invalid parameter (negative number): " + number);
170     }
171
172     /**
173      * Initiates the sub-workflow.
174      */
175     protected void executeExecuteCalculusSubFlow(Subflow s) throws Exception {
176        System.out.println("Executing: " + number);
177        s.fill("number", number);
178        s.setPerformer(((FactorialAgent) myAgent).getCalculatorAgent().getLocalName());
179        performSubflow(s);
180        number -= 2;
181     }
182
183     /**
184      * Check if there is more elements to calculate.
185      */
186     protected boolean checkhasMore() throws Exception {
187        return number > 1L;
188     }
189
190     /**
191      * <br>
192      */
193     protected void executeFactorialWorkflowSubflowJoinActivity1(SubflowList ss)
194           throws Exception {
195        List<Subflow> ls;
196        ls = (List<Subflow>) ss.get(EXECUTECALCULUSSUBFLOW_ACTIVITY);
197
198        result = 1L;
199        for (Subflow tmpS : ls) {
200           Long flowResult = (Long) tmpS.extract("result");
201           result *= flowResult;
202        }
203     }
```

MultiplyWorkflow.java Source-code of Multiplication Workflow

```
1  package fr.utc.agent.workflow;
2
3  import com.tilab.wade.performer.layout.ActivityLayout;
4  import com.tilab.wade.performer.layout.WorkflowLayout;
5  import com.tilab.wade.performer.CodeExecutionBehaviour;
6  import com.tilab.wade.performer.FormalParameter;
7  import com.tilab.wade.performer.WorkflowBehaviour;
8
9  @WorkflowLayout(activities = { @ActivityLayout(label = "Perform the calculus", position = "(265,11
10 public class MultiplyWorkflow extends WorkflowBehaviour {
11
12    public static final String EXECUTECALCULUS_ACTIVITY = "ExecuteCalculus";
13
```

```
14      private static final long serialVersionUID = 6554376328674062600L;
15
16      @FormalParameter
17      private Long number;
18
19      @FormalParameter(mode=FormalParameter.OUTPUT)
20      private Long result;
21
22      @SuppressWarnings("unused")
23      private void defineActivities() {
24          CodeExecutionBehaviour executeCalculusActivity = new CodeExecutionBehaviour(
25                  EXECUTECALCULUS_ACTIVITY, this);
26          registerActivity(executeCalculusActivity, INITIAL_AND_FINAL);
27      }
28
29      /**
30       * Execute the activity.
31       */
32      protected void executeExecuteCalculus() throws Exception {
33
34          Long subsequent = number == 1 ? 1 : number - 1;
35
36          result = number * subsequent;
37      }
38  }
```