UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique

OMAS v8 - User Manual

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
November 2010

Warning

This document contains the documentation for using the OMAS platform. It applies to the OMAS v8 release. However, some of the chapters were written with previous releases and may present some slight differences with OMAS v8.

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	November 10	Barthès	First Issue

Contents

Chapter 1

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.

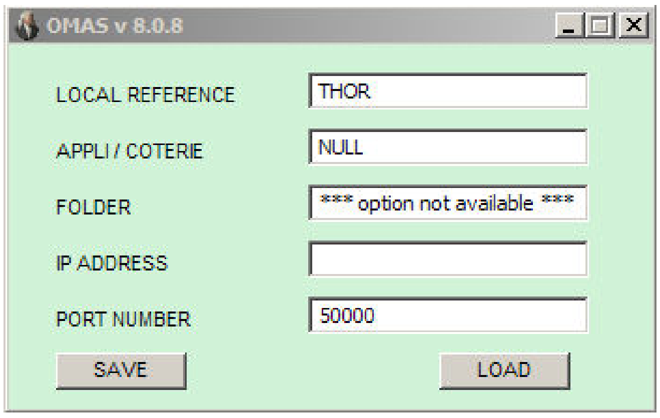
1.1 Tutorial 1: Simple Introduction to the OMAS Platform

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

1.1.1 How to Get Started

First, one has to load the platform.

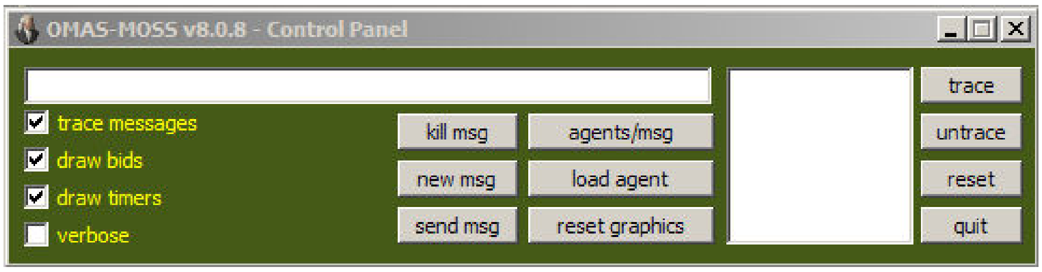
To load the OMAS, simply launch the LISP environment (Allegro CL with IDE), then using the "Load " entry of the "File" menu load the file named "load-omas-moss.fasl" or "load-omas-moss.lisp" located in your OMAS-MOSS folder.



LOCAL REFERENCE	THOR
APPLI / COTERIE	NULL
FOLDER	*** option not available ***
IP ADDRESS	
PORT NUMBER	50000

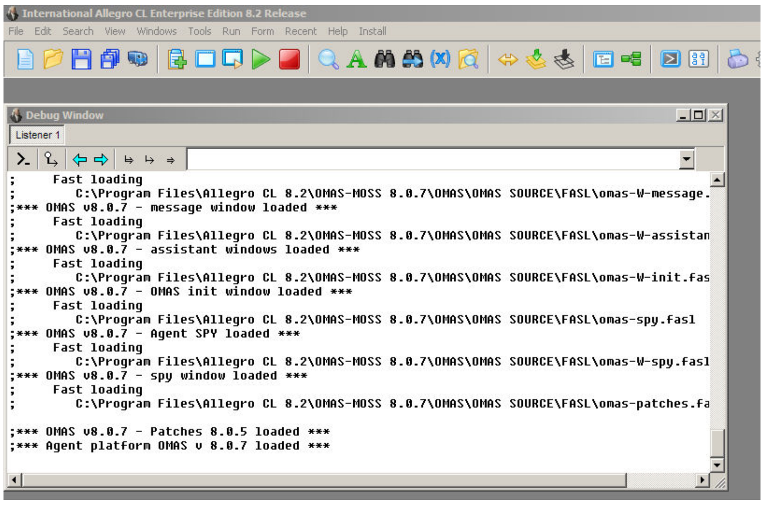
Figure 1.1: Initial OMAS menu

After the file is loaded (ignore the various warning messages displayed while loading), a menu appears as shown Fig.1.1. Clicking the LOAD button produces the OMAS Control Panel (Fig.1.2) on the top left part of the screen. The control panel allows controlling and tracing what is happening with the agents.



- [✓] trace messages
- [✓] draw bids
- [✓] draw timers
- [ ] verbose

Figure 1.2: OMAS Control Panel


International Allegro CL Enterprise Edition 8.2 Release
File Edit Search View Windows Tools Run Form Recent Help Install
Debug Window
Listener 1

```
;     Fast loading
;        C:\Program Files\Allegro CL 8.2\OMAS-MOSS 8.0.7\OMAS\OMAS SOURCE\FASL\omas-W-message.
;*** OMAS v8.0.7 - message window loaded ***
;     Fast loading
;        C:\Program Files\Allegro CL 8.2\OMAS-MOSS 8.0.7\OMAS\OMAS SOURCE\FASL\omas-W-assistan
;*** OMAS v8.0.7 - assistant windows loaded ***
;     Fast loading
;        C:\Program Files\Allegro CL 8.2\OMAS-MOSS 8.0.7\OMAS\OMAS SOURCE\FASL\omas-W-init.fas
;*** OMAS v8.0.7 - OMAS init window loaded ***
;     Fast loading
;        C:\Program Files\Allegro CL 8.2\OMAS-MOSS 8.0.7\OMAS\OMAS SOURCE\FASL\omas-spy.fasl
;*** OMAS v8.0.7 - Agent SPY loaded ***
;     Fast loading
;        C:\Program Files\Allegro CL 8.2\OMAS-MOSS 8.0.7\OMAS\OMAS SOURCE\FASL\omas-W-spy.fasl
;*** OMAS v8.0.7 - spy window loaded ***
;     Fast loading
;        C:\Program Files\Allegro CL 8.2\OMAS-MOSS 8.0.7\OMAS\OMAS SOURCE\FASL\omas-patches.fa

;*** OMAS v8.0.7 - Patches 8.0.5 loaded ***
;*** Agent platform OMAS v 8.0.7 loaded ***
```

Figure 1.3: The Lisp Listener (ACL Debug Window) after OMAS is loaded (may not look exactly as this one)

1.1.2 Creating an Agent

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

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

The agent named `TEST`[2] has been created and is 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.

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

1.1.3 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:

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

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
```

[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.

[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 (:).

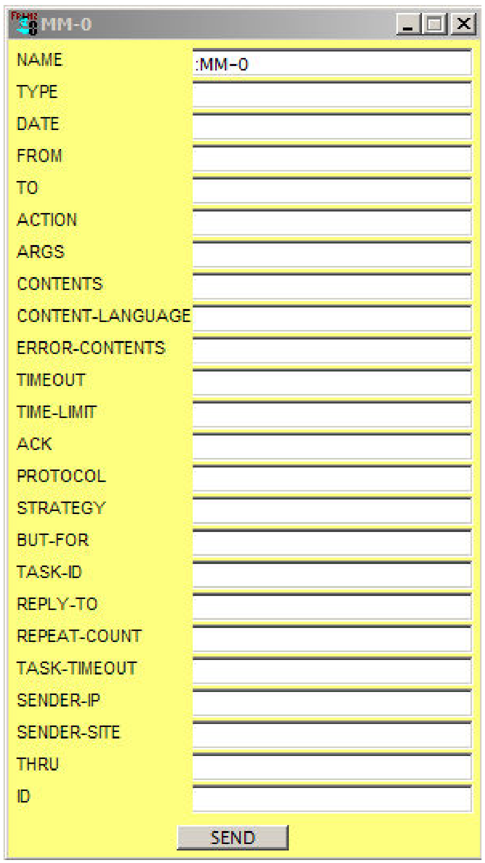
The two first arguments (agent and message) are required, the last one (arg) is the first argument of the message received by the agent; agent is the current agent TEST, message refers to the request message that was just received when the function is executed.

More details: Note the use of the `static-exit` function that 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.

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 answer.

1.1.4 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.1.4.



NAME	:MM-0
TYPE	
DATE	
FROM	
TO	
ACTION	
ARGS	
CONTENTS	
CONTENT-LANGUAGE	
ERROR-CONTENTS	
TIMEOUT	
TIME-LIMIT	
ACK	
PROTOCOL	
STRATEGY	
BUT-FOR	
TASK-ID	
REPLY-TO	
REPEAT-COUNT	
TASK-TIMEOUT	
SENDER-IP	
SENDER-SITE	
THRU	
ID	

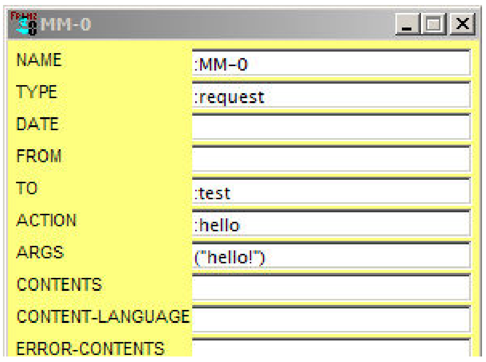
Figure 1.4: Message editing window

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.



NAME	:MM-0
TYPE	:request
DATE	
FROM	
TO	:test
ACTION	:hello
ARGS	("hello!")
CONTENTS	
CONTENT-LANGUAGE	
ERROR-CONTENTS	

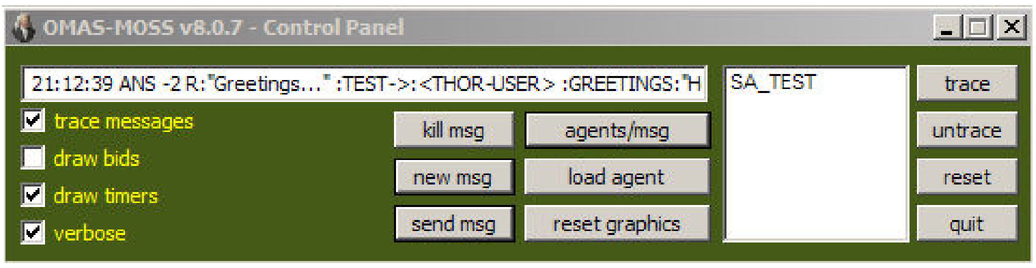
Figure 1.5: Filling the message window for the agent TEST

We then click the "send" button at the bottom of the message window. Nothing seems to happen. In fact TEST has received the message and answered it. Since the answer is to the system, the answer message is displayed in the small horizontal window of the control panel (Fig.1.6). 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!"

Although this is informative, it does not allow us to follow exactly what was going on. To do so, we must use some of the features of the environment handled from the control panel.



```
21:12:39 ANS -2 R:"Greetings..." :TEST->:<THOR-USER> :GREETINGS:"H
```

- [✓] trace messages
- [ ] draw bids
- [✓] draw timers
- [✓] verbose

SA_TEST

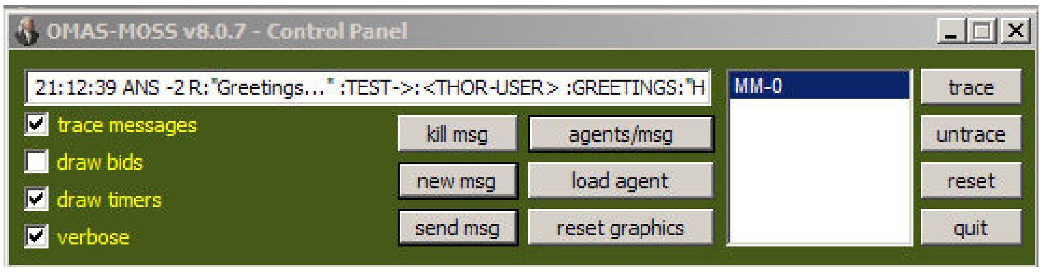
Figure 1.6: Panel showing returned message

1.1.5 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 TEST appears). If we click it, then the MM-0 message appears replacing the TEST agent (Fig.1.7). If we click it again the TEST agent reappears.

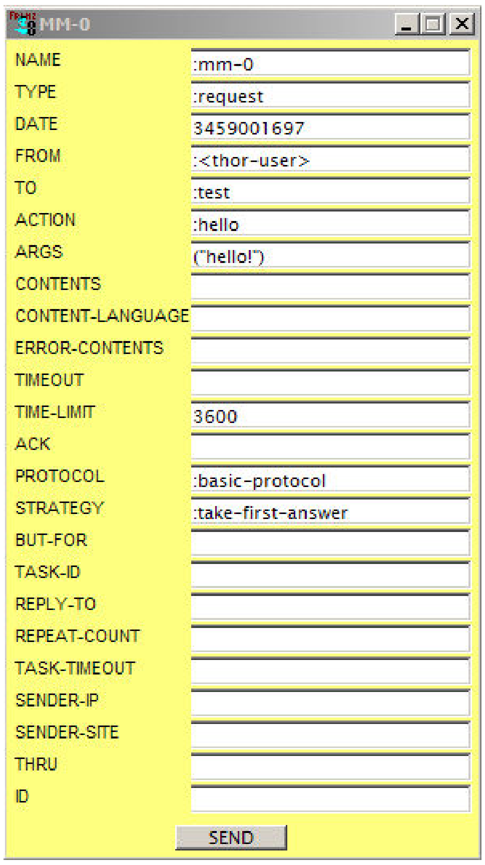


```
21:12:39 ANS -2 R:"Greetings..." :TEST->:<THOR-USER> :GREETINGS:"H
```

- [✓] trace messages
- [ ] draw bids
- [✓] draw timers
- [✓] verbose

MM-0

Figure 1.7: Displaying messages (MM-0) in the scrolling list pane



NAME	:mm-0
TYPE	:request
DATE	3459001697
FROM	:<thor-user>
TO	:test
ACTION	:hello
ARGS	("hello!")
CONTENTS	
CONTENT-LANGUAGE	
ERROR-CONTENTS	
TIMEOUT	
TIME-LIMIT	3600
ACK	
PROTOCOL	:basic-protocol
STRATEGY	:take-first-answer
BUT-FOR	
TASK-ID	
REPLY-TO	
REPEAT-COUNT	
TASK-TIMEOUT	
SENDER-IP	
SENDER-SITE	
THRU	
ID	

Figure 1.8: Message edit window

If we double-click on MM-0, then the edit window appears (Fig.1.8).

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 TEST, a special agent window appears right underneath the control panel as shown Fig.1.9. The content of this window will be detailed later.

Showing the Exchange of Messages

To display the exchange of messages in the environment, one must click on the "reset-graphics" button. OMAS then displays a window where messages are traced, as shown Fig.1.10.

Now we can send the message MM-0 again by selecting the message in the message list and clicking

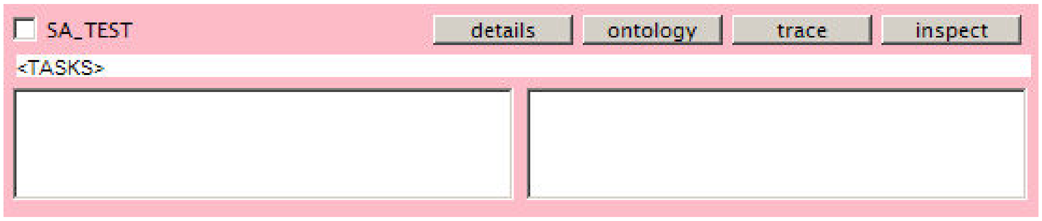


Figure 1.9: Agent window for TEST

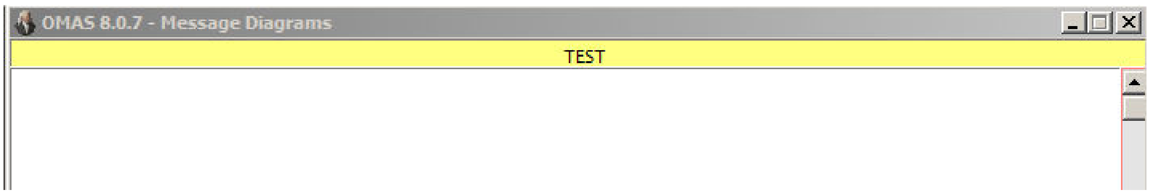
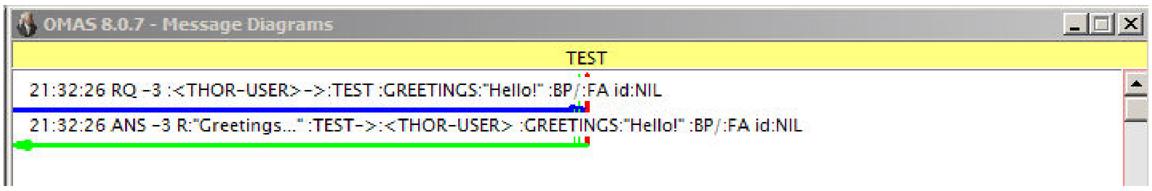


Figure 1.10: Graphics trace window

the "send msg" button, and watch the result in the Message Diagram window, where two messages appear:

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 window. This is where the user lives!?



```
21:32:26 RQ -3 :<THOR-USER>->:TEST :GREETINGS:"Hello!" :BP/:FA id:NIL
21:32:26 ANS -3 R:"Greetings..." :TEST->:<THOR-USER> :GREETINGS:"Hello!" :BP/:FA id:NIL
```

Figure 1.11: Message trace

Clicking the "reset graphics" button resets the "Message Diagram" window.

Sending a Message Again

To send a message again, one can either double-click on the message, and click on the "send" button 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.

Buttons 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.

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).

1.1.6 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 the various scenarios can be exercised.

To load the factorial demo, load the factorial.lisp file from the application folder of the OMAS-Vxx folder. This loads the demo file and starts the four agents FAC, MUL-1, MUL-2 and MUL-3 agents. At the same time several messages are predefined and loaded: DF, DFWTL, GF, CNF. Clicking on the "agents/msg" button displays the agents (Fig.2.1), then the messages (Fig.2.8).

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.1.14.

The result, 24, can be seen in the text of the last message and appears in the answer line of the control panel.

In order to implement this 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,

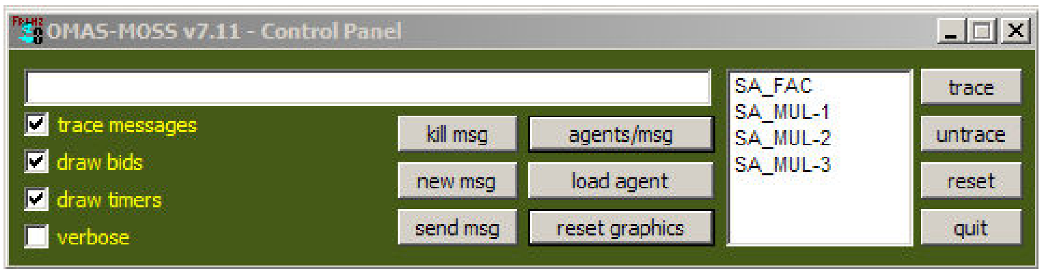


Figure 1.12: Agents from the factorial application

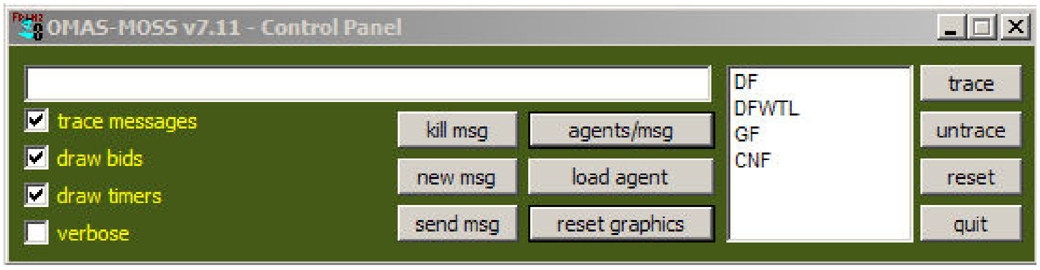
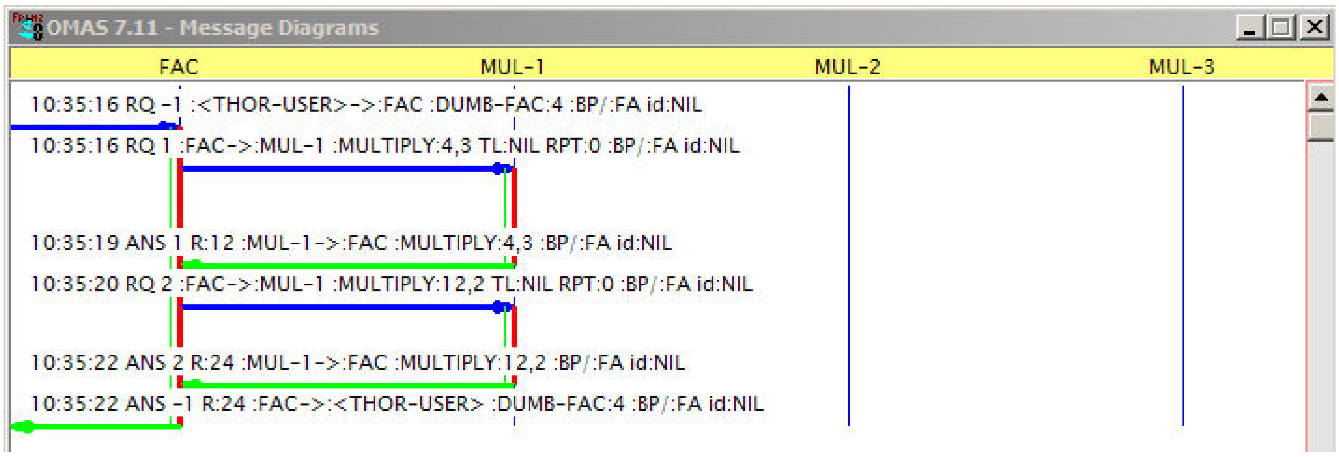


Figure 1.13: Messages for the factorial application



```
10:35:16 RQ -1 :<THOR-USER>->:FAC :DUMB-FAC:4 :BP/:FA id:NIL
10:35:16 RQ 1 :FAC->:MUL-1 :MULTIPLY:4,3 TL:NIL RPT:0 :BP/:FA id:NIL
10:35:19 ANS 1 R:12 :MUL-1->:FAC :MULTIPLY:4,3 :BP/:FA id:NIL
10:35:20 RQ 2 :FAC->:MUL-1 :MULTIPLY:12,2 TL:NIL RPT:0 :BP/:FA id:NIL
10:35:22 ANS 2 R:24 :MUL-1->:FAC :MULTIPLY:12,2 :BP/:FA id:NIL
10:35:22 ANS -1 R:24 :FAC->:<THOR-USER> :DUMB-FAC:4 :BP/:FA id:NIL
```

Figure 1.14: Computing 4! using the dumb strategy

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.

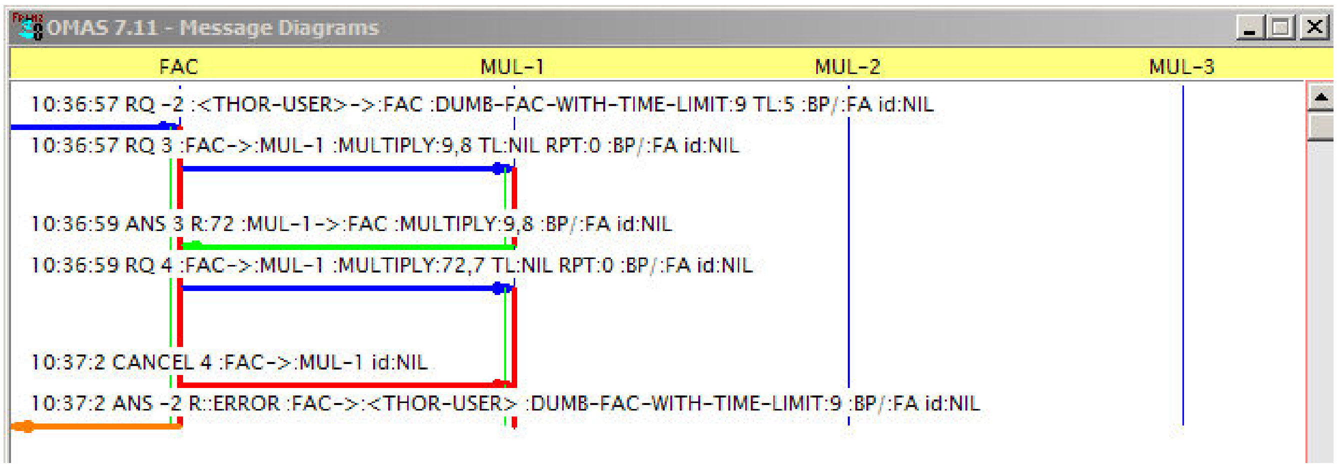
The dynamic part of the factorial receives an answer and must decide whether the computation is finished or not. When it is not finished, it sends another multiply to MUL-1. When the computation is finished it calls the dynamic-exit function.

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

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

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.1.15). The time limit is set as a parameter of the message sent to the agent FAC.



```
10:36:57 RQ -2 :<THOR-USER>->:FAC :DUMB-FAC-WITH-TIME-LIMIT:9 TL:5 :BP/:FA id:NIL
10:36:57 RQ 3 :FAC->:MUL-1 :MULTIPLY:9,8 TL:NIL RPT:0 :BP/:FA id:NIL
10:36:59 ANS 3 R:72 :MUL-1->:FAC :MULTIPLY:9,8 :BP/:FA id:NIL
10:36:59 RQ 4 :FAC->:MUL-1 :MULTIPLY:72,7 TL:NIL RPT:0 :BP/:FA id:NIL
10:37:2 CANCEL 4 :FAC->:MUL-1 id:NIL
10:37:2 ANS -2 R::ERROR :FAC->:<THOR-USER> :DUMB-FAC-WITH-TIME-LIMIT:9 :BP/:FA id:NIL
```

Figure 1.15: 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:

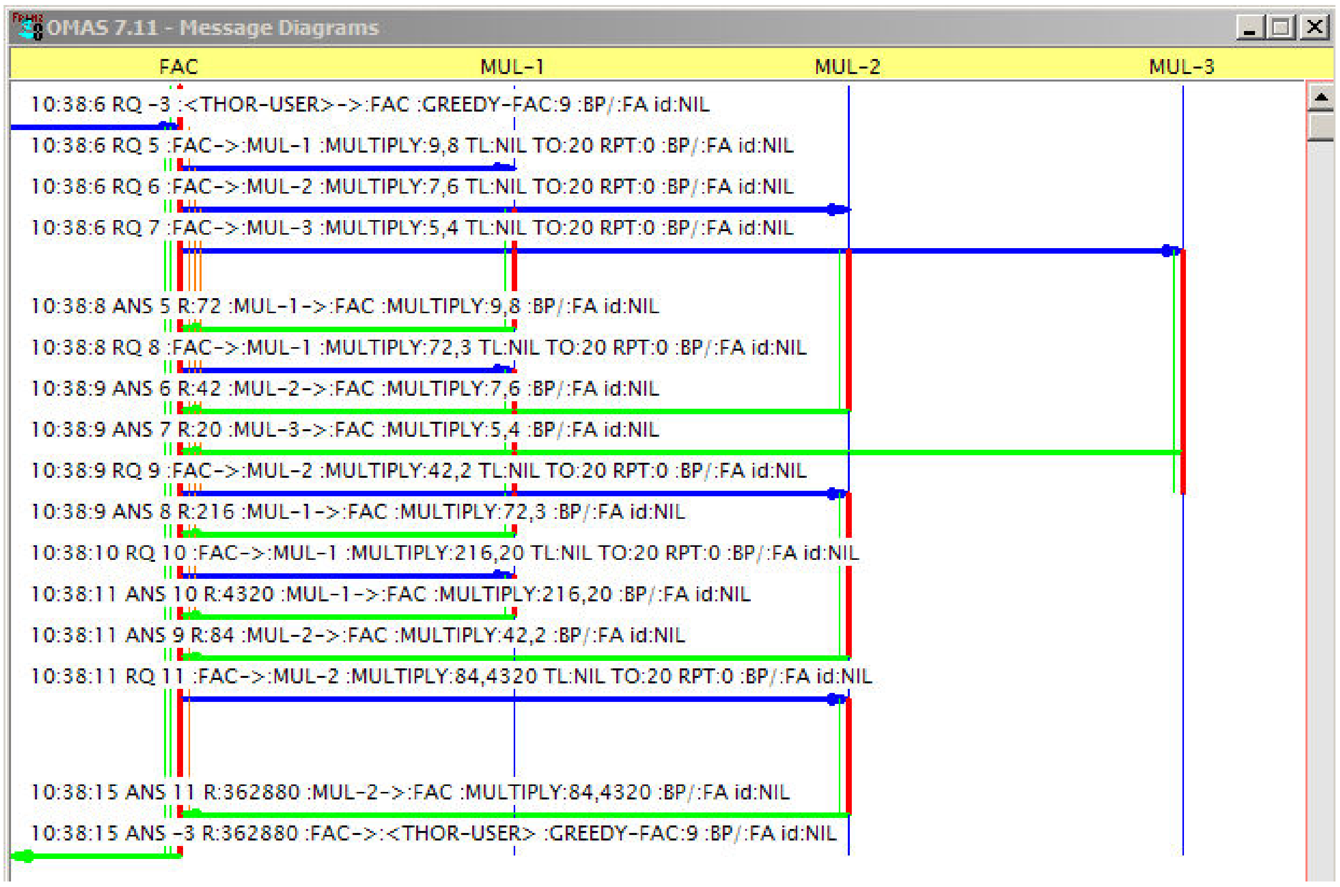
```
(omas::defskill :dumb-fac-with-time-limit :FAC
  :static-fcn static-dumb-fac
  :dynamic-fcn dynamic-dumb-fac
  :time-limit-fcn time-limit-dumb-fac
  )
with the corresponding code:

(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.1.16 for the computation of fac(9).

The code for the skills is the following:

```
(defskill greedy-factorial :FAC
```



```
10:38:6 RQ -3 :<THOR-USER>->:FAC :GREEDY-FAC:9 :BP/:FA id:NIL
10:38:6 RQ 5 :FAC->:MUL-1 :MULTIPLY:9,8 TL:NIL TO:20 RPT:0 :BP/:FA id:NIL
10:38:6 RQ 6 :FAC->:MUL-2 :MULTIPLY:7,6 TL:NIL TO:20 RPT:0 :BP/:FA id:NIL
10:38:6 RQ 7 :FAC->:MUL-3 :MULTIPLY:5,4 TL:NIL TO:20 RPT:0 :BP/:FA id:NIL
10:38:8 ANS 5 R:72 :MUL-1->:FAC :MULTIPLY:9,8 :BP/:FA id:NIL
10:38:8 RQ 8 :FAC->:MUL-1 :MULTIPLY:72,3 TL:NIL TO:20 RPT:0 :BP/:FA id:NIL
10:38:9 ANS 6 R:42 :MUL-2->:FAC :MULTIPLY:7,6 :BP/:FA id:NIL
10:38:9 ANS 7 R:20 :MUL-3->:FAC :MULTIPLY:5,4 :BP/:FA id:NIL
10:38:9 RQ 9 :FAC->:MUL-2 :MULTIPLY:42,2 TL:NIL TO:20 RPT:0 :BP/:FA id:NIL
10:38:9 ANS 8 R:216 :MUL-1->:FAC :MULTIPLY:72,3 :BP/:FA id:NIL
10:38:10 RQ 10 :FAC->:MUL-1 :MULTIPLY:216,20 TL:NIL TO:20 RPT:0 :BP/:FA id:NIL
10:38:11 ANS 10 R:4320 :MUL-1->:FAC :MULTIPLY:216,20 :BP/:FA id:NIL
10:38:11 ANS 9 R:84 :MUL-2->:FAC :MULTIPLY:42,2 :BP/:FA id:NIL
10:38:11 RQ 11 :FAC->:MUL-2 :MULTIPLY:84,4320 TL:NIL TO:20 RPT:0 :BP/:FA id:NIL
10:38:15 ANS 11 R:362880 :MUL-2->:FAC :MULTIPLY:84,4320 :BP/:FA id:NIL
10:38:15 ANS -3 R:362880 :FAC->:<THOR-USER> :GREEDY-FAC:9 :BP/:FA id:NIL
```

Figure 1.16: Computing 9! using the Greedy strategy

```
  :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.1.16 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)
```

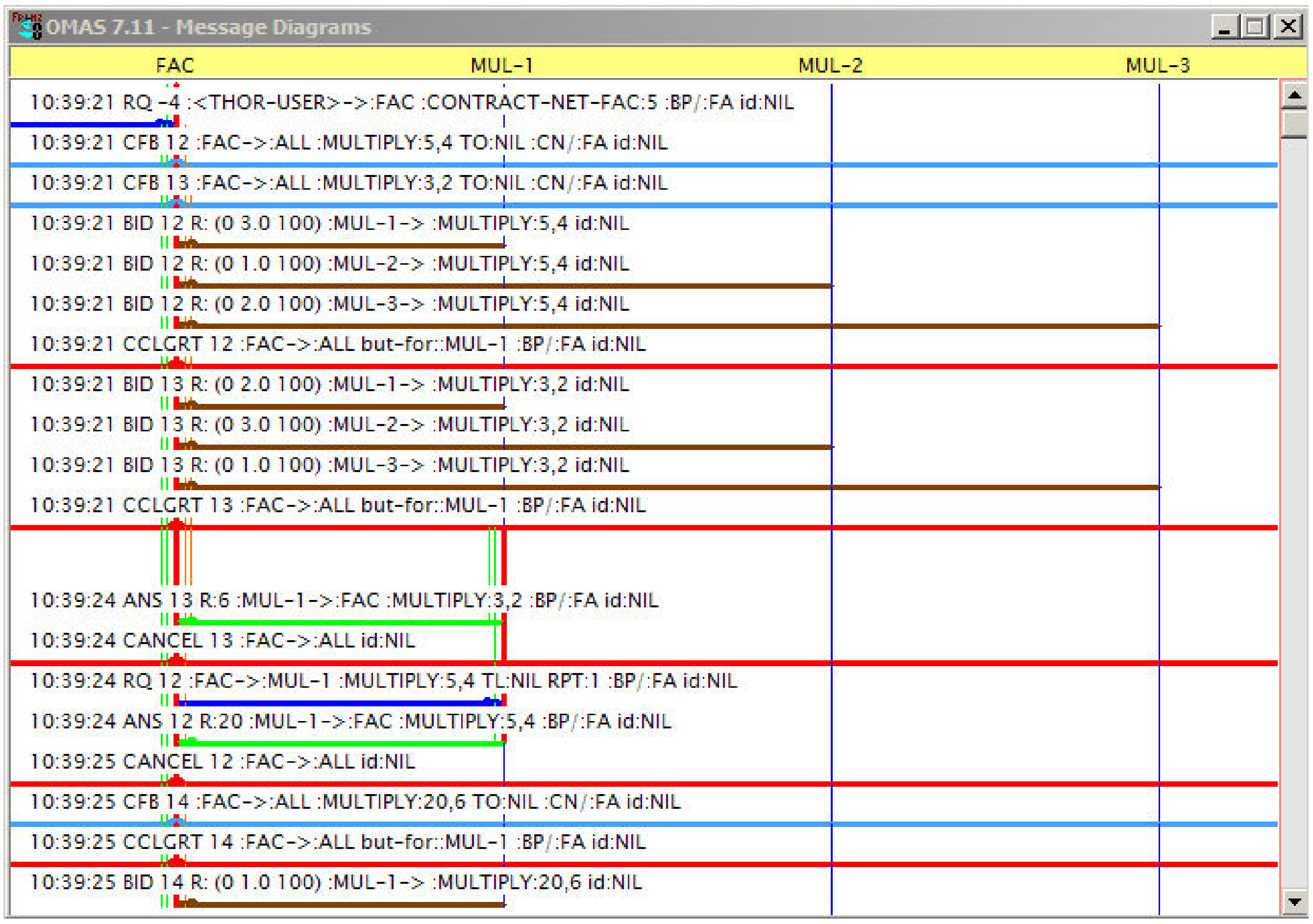
Contract Net Strategy

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.1.17) and MULTIPLY agents answer the bids offering services (brown messages). 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.1.18 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:

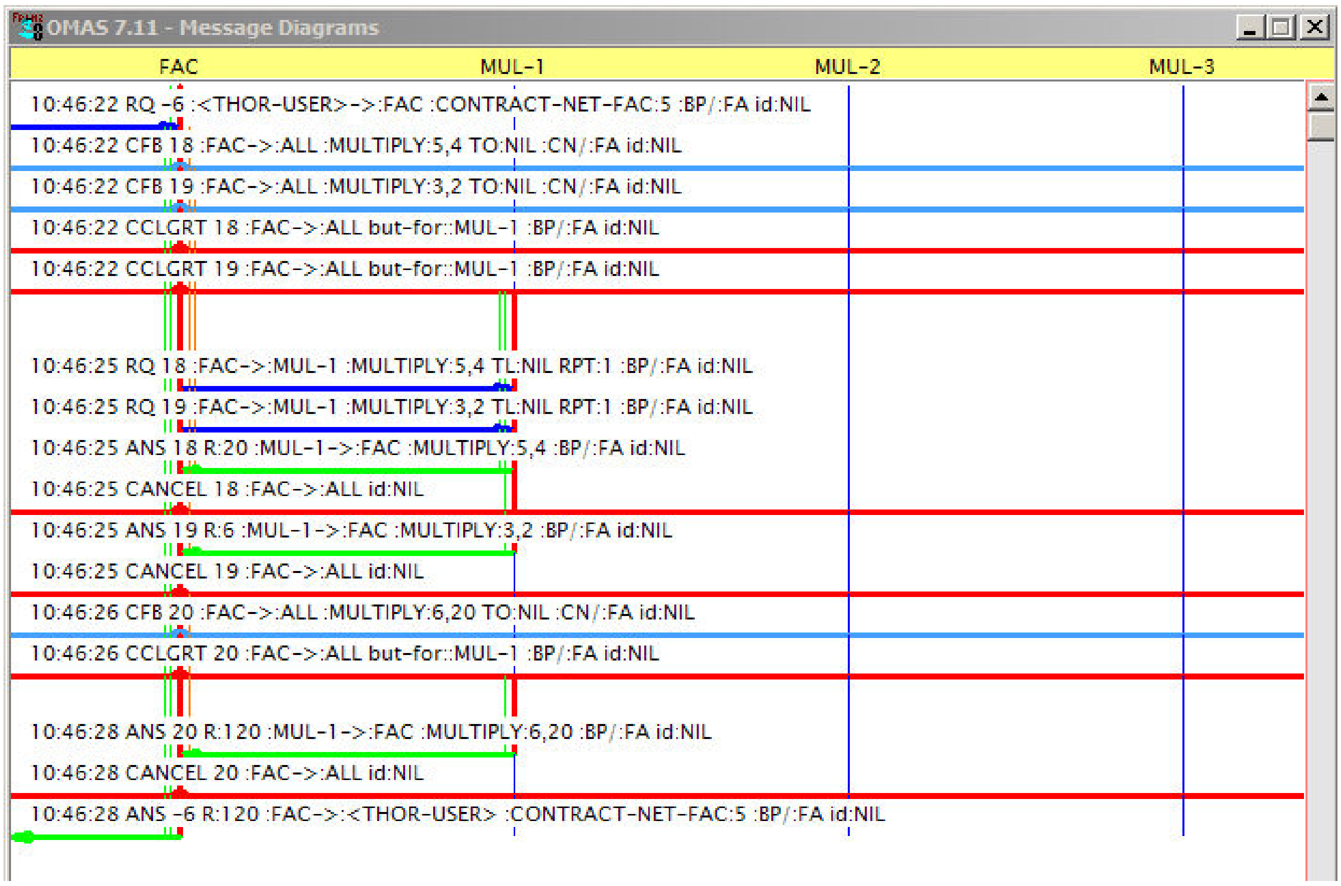
```
(defskill :contract-net-fac :fac
  :static-fcn static-contract-net-fac
  :dynamic-fcn dynamic-contract-net-fac
  )

(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
          ;; create n/2 tasks for distribution to the multiplying agents
```



```
10:39:21 RQ -4 :<THOR-USER>->:FAC :CONTRACT-NET-FAC:5 :BP/:FA id:NIL
10:39:21 CFB 12 :FAC->:ALL :MULTIPLY:5,4 TO:NIL :CN/:FA id:NIL
10:39:21 CFB 13 :FAC->:ALL :MULTIPLY:3,2 TO:NIL :CN/:FA id:NIL
10:39:21 BID 12 R: (0 3.0 100) :MUL-1-> :MULTIPLY:5,4 id:NIL
10:39:21 BID 12 R: (0 1.0 100) :MUL-2-> :MULTIPLY:5,4 id:NIL
10:39:21 BID 12 R: (0 2.0 100) :MUL-3-> :MULTIPLY:5,4 id:NIL
10:39:21 CCLGRT 12 :FAC->:ALL but-for::MUL-1 :BP/:FA id:NIL
10:39:21 BID 13 R: (0 2.0 100) :MUL-1-> :MULTIPLY:3,2 id:NIL
10:39:21 BID 13 R: (0 3.0 100) :MUL-2-> :MULTIPLY:3,2 id:NIL
10:39:21 BID 13 R: (0 1.0 100) :MUL-3-> :MULTIPLY:3,2 id:NIL
10:39:21 CCLGRT 13 :FAC->:ALL but-for::MUL-1 :BP/:FA id:NIL
10:39:24 ANS 13 R:6 :MUL-1->:FAC :MULTIPLY:3,2 :BP/:FA id:NIL
10:39:24 CANCEL 13 :FAC->:ALL id:NIL
10:39:24 RQ 12 :FAC->:MUL-1 :MULTIPLY:5,4 TL:NIL RPT:1 :BP/:FA id:NIL
10:39:24 ANS 12 R:20 :MUL-1->:FAC :MULTIPLY:5,4 :BP/:FA id:NIL
10:39:25 CANCEL 12 :FAC->:ALL id:NIL
10:39:25 CFB 14 :FAC->:ALL :MULTIPLY:20,6 TO:NIL :CN/:FA id:NIL
10:39:25 CCLGRT 14 :FAC->:ALL but-for::MUL-1 :BP/:FA id:NIL
10:39:25 BID 14 R: (0 1.0 100) :MUL-1-> :MULTIPLY:20,6 id:NIL
```

Figure 1.17: Computing 5! using a Contract-Net strategy

```
          ;; 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.
```



```
10:46:22 RQ -6 :<THOR-USER>->:FAC :CONTRACT-NET-FAC:5 :BP/:FA id:NIL
10:46:22 CFB 18 :FAC->:ALL :MULTIPLY:5,4 TO:NIL :CN/:FA id:NIL
10:46:22 CFB 19 :FAC->:ALL :MULTIPLY:3,2 TO:NIL :CN/:FA id:NIL
10:46:22 CCLGRT 18 :FAC->:ALL but-for::MUL-1 :BP/:FA id:NIL
10:46:22 CCLGRT 19 :FAC->:ALL but-for::MUL-1 :BP/:FA id:NIL
10:46:25 RQ 18 :FAC->:MUL-1 :MULTIPLY:5,4 TL:NIL RPT:1 :BP/:FA id:NIL
10:46:25 RQ 19 :FAC->:MUL-1 :MULTIPLY:3,2 TL:NIL RPT:1 :BP/:FA id:NIL
10:46:25 ANS 18 R:20 :MUL-1->:FAC :MULTIPLY:5,4 :BP/:FA id:NIL
10:46:25 CANCEL 18 :FAC->:ALL id:NIL
10:46:25 ANS 19 R:6 :MUL-1->:FAC :MULTIPLY:3,2 :BP/:FA id:NIL
10:46:25 CANCEL 19 :FAC->:ALL id:NIL
10:46:26 CFB 20 :FAC->:ALL :MULTIPLY:6,20 TO:NIL :CN/:FA id:NIL
10:46:26 CCLGRT 20 :FAC->:ALL but-for::MUL-1 :BP/:FA id:NIL
10:46:28 ANS 20 R:120 :MUL-1->:FAC :MULTIPLY:6,20 :BP/:FA id:NIL
10:46:28 CANCEL 20 :FAC->:ALL id:NIL
10:46:28 ANS -6 R:120 :FAC->:<THOR-USER> :CONTRACT-NET-FAC:5 :BP/:FA id:NIL
```

Figure 1.18: Computing 5! with a Contract-Net strategy hiding bids

```
   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 mechanism is taken care of by the OMAS platform.

1.1.7 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 defined as follows:

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

Alternatively, they can be constructed interactively 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. One of the differences between OMAS and other multi-agent platforms like JADE is that 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.

File Organization

In this document applications are built in the shared cg-user space. 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 1.2.

1.1.8 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 each different machine. See Section 1.2.7.

1.2 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.

1.2.1 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 **graphical 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.

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.

Fig1.19 shows an example of coterie. Agents in the oval section are located in the same Lisp environment on the user's machine (although this is 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.

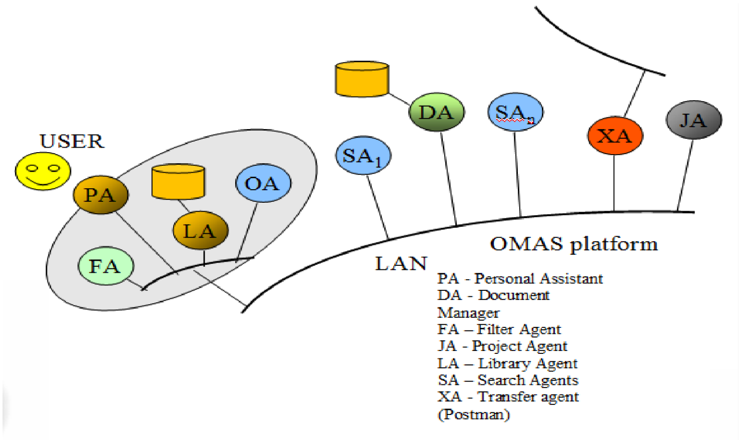


Figure 1.19: Example of OMAS coterie

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.

1.2.2 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 distance 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 1.2.3 to 1.2.5 develop a small example illustrating the use of a service agent and of a personal assistant agent.

1.2.3 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

1.2.4 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-TIME 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[5], e.g. Notepad++ or Emacs. We found that the ACL editor introduces spurious characters in UTF-8 encoded files. The MCL editor however can be used directly.
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 definiting the agent.

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

(defpackage :CALENDAR (:use :moss :omas :cl #+MCL :ccl))
(in-package :CALENDAR)

;;;===========================================================================
;;;
;;;                              defining the agent
;;;
;;;===========================================================================
```

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

```
;;; :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 :redefine t)
...
```

The GET-DATA Skill

The skill section is included in comments as follows:

```
;;; ==============================================================================
;;;                                 Skill  XXX
;;; ==============================================================================
#|
(defskill :XXX :CALENDAR
  :static-fcn static-XXX
  :dynamic-fcn dynamic-XXX
  :time-limit-fcn time-limit-XXX
  :timeout-handler timeout-handler-XXX
  :preconditions preconditions-XXX
  :select-best-answer-fcn select-best-answer-XXX
  :acknowledge-fcn acknowledge-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 time-limit-xxx (agent message message)
    "documentation"
  )

(defun timeout-handler-xxx (agent message)
    "documentation"
  )

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

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

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

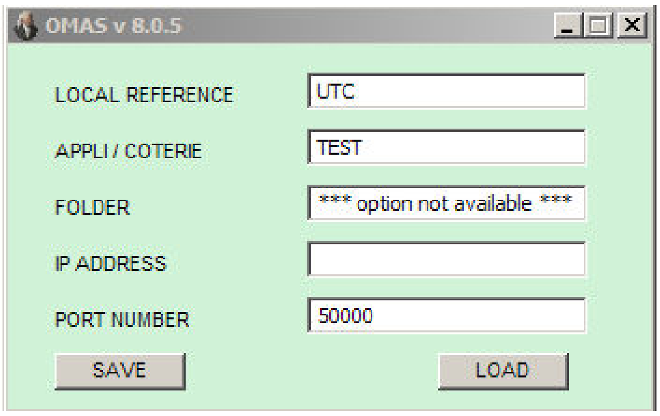
Loading the Application

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.1.20). The name of our folder, UTC-TEST, has two parts, the first one, UTC, is a reference to the local coterie, the second part, TEST, is the name of the application.



LOCAL REFERENCE: UTC
APPLI / COTERIE: TEST
FOLDER: *** option not available ***
IP ADDRESS:
PORT NUMBER: 50000

Figure 1.20: OMAS Initial Menu

We can save the reference by clicking the SAVE button and start OMAS by clicking LOAD. 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.1.20 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).

1.2.5 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

1.2.6 Adding OSCAR to the Application

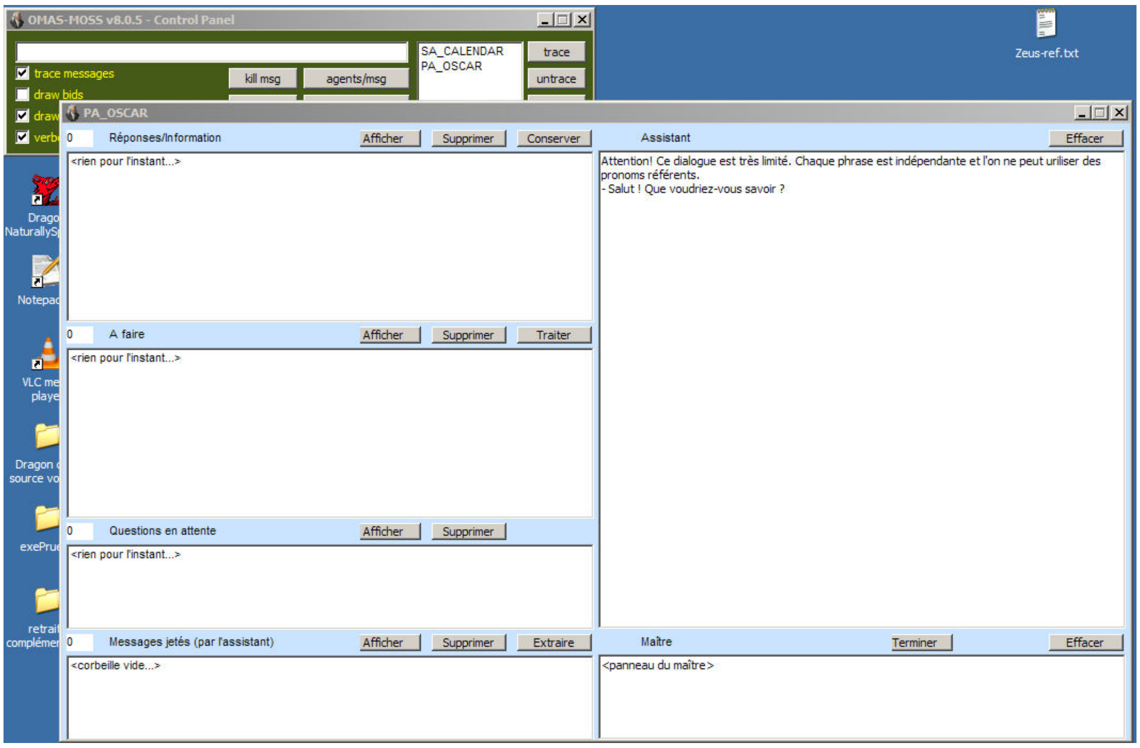
To do so:

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.1.21).



<rien pour l'instant...>

Attention! Ce dialogue est très limité. Chaque phrase est indépendante et l'on ne peut uriliser des pronoms référents.
- Salut ! Que voudriez-vous savoir ?

<corbeille vide...>

<panneau du maître>

Figure 1.21: OMAS Initial Menu

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
- 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. The conversation 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
    :exec (omas::assistant-display-text *current-agent*
                (omas::answer *current-agent*))
: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.

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

1.2.7 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 by program. 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 LC 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).

1.3 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 4). Features are introduced roughly in order of increasing complexity.

1.3.1 IDE Features

How to use the IDE is described in Chapter 2.

1.3.2 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. This is done 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.

1.3.3 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.

1.3.4 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.

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.

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

1.3.5 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.

1.3.6 Using Broadcast

We use broacast 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 (environment) 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)
```

1.3.7 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 doen 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.

1.3.8 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.

1.3.9 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 **??**.

1.3.10 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 4.3.

1.3.11 Defining Dialogs

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

1.3.12 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. They are presented in a different document.

Chapter 2

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 Tutorial 1.

2.1 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 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

2.1.1 Examining Agents

Viewing the List of Agents

The agents of the application appear in the right pane of the control panel. Fig.2.1 displays the four agents of the factorial example. Note that the name of each agent is prefixed by SA_ for Service Agent.

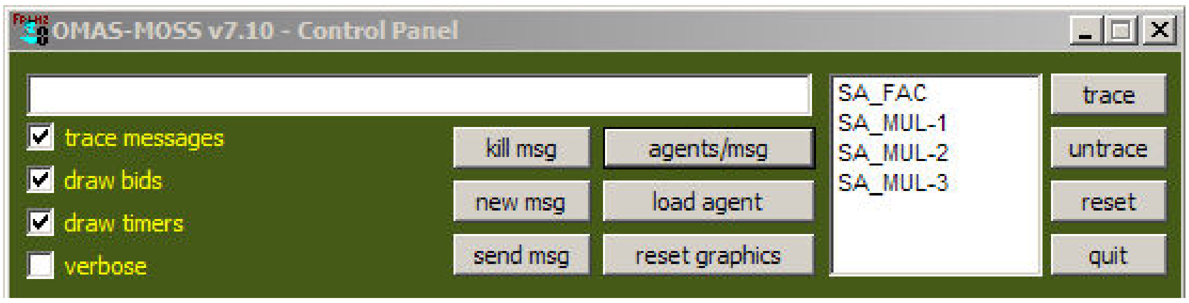
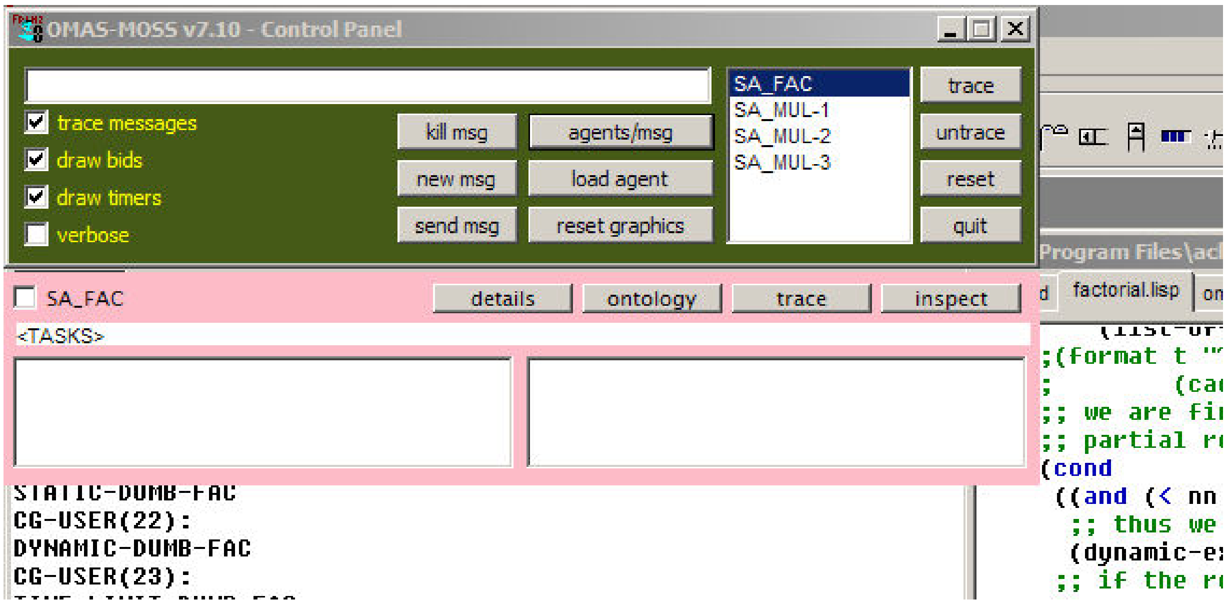


Figure 2.1: Agents from the factorial application

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.2.2).



```
STATIC-DUMB-FAC
CG-USER(22):
DYNAMIC-DUMB-FAC
CG-USER(23):
```

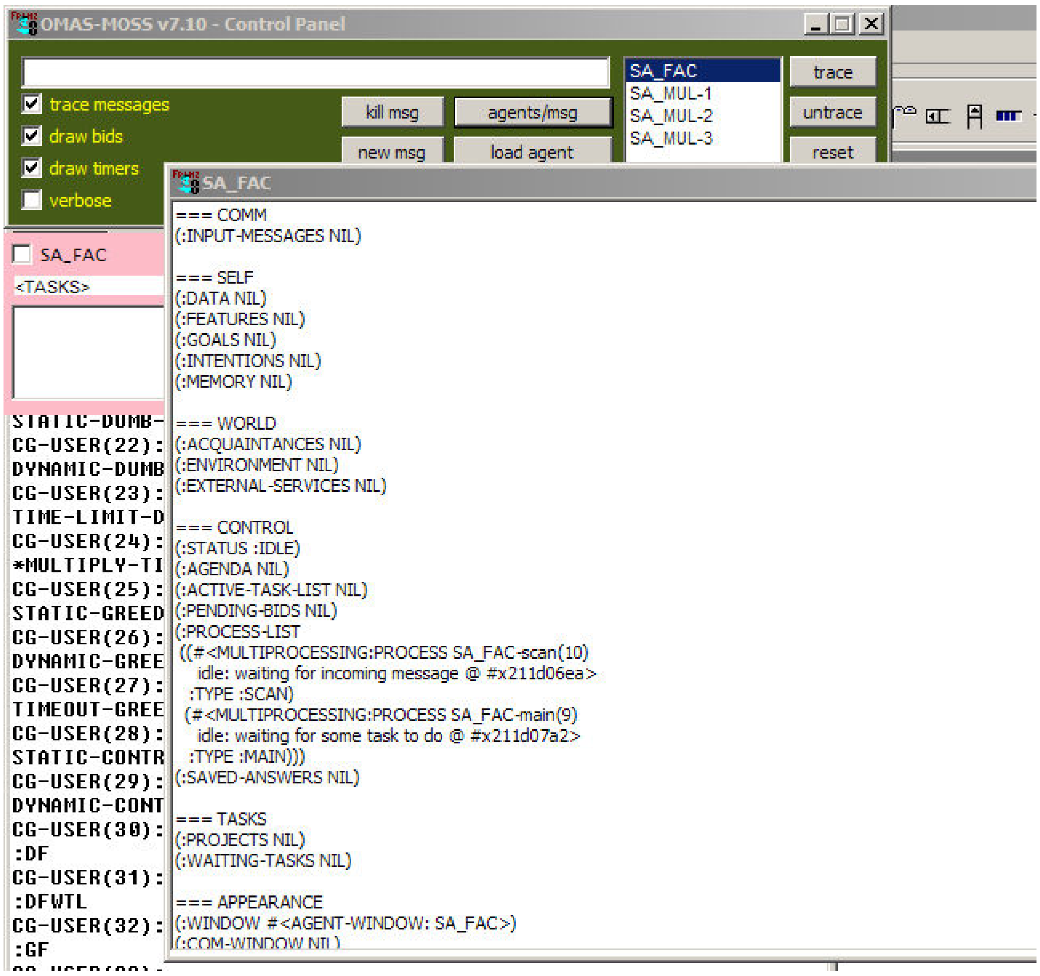
```
;(format t "
;       (ca
;; we are fi
;; partial r
(cond
 ((and (< nn
  ;; thus we
  (dynamic-e
 ;; if the r
```

Figure 2.2: Agent window

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.2.3)

- **ontology** that displays the agent ontology (none in the case of the factorial example)
- **trace** for tracing the agent
- **inspect** for opening a Lisp inspection window on the agent structure (for Lisp specialists)



```
STATIC-DUMB-
CG-USER(22):
DYNAMIC-DUMB
CG-USER(23):
TIME-LIMIT-D
CG-USER(24):
*MULTIPLY-TI
CG-USER(25):
STATIC-GREED
CG-USER(26):
DYNAMIC-GREE
CG-USER(27):
TIMEOUT-GREE
CG-USER(28):
STATIC-CONTR
CG-USER(29):
DYNAMIC-CONT
CG-USER(30):
:DF
CG-USER(31):
:DFWTL
CG-USER(32):
:GF
```

```
=== COMM
(:INPUT-MESSAGES NIL)

=== SELF
(:DATA NIL)
(:FEATURES NIL)
(:GOALS NIL)
(:INTENTIONS NIL)
(:MEMORY NIL)

=== WORLD
(:ACQUAINTANCES NIL)
(:ENVIRONMENT NIL)
(:EXTERNAL-SERVICES NIL)

=== CONTROL
(:STATUS :IDLE)
(:AGENDA NIL)
(:ACTIVE-TASK-LIST NIL)
(:PENDING-BIDS NIL)
(:PROCESS-LIST
 ((#<MULTIPROCESSING:PROCESS SA_FAC-scan(10)
    idle: waiting for incoming message @ #x211d06ea>
   :TYPE :SCAN)
  (#<MULTIPROCESSING:PROCESS SA_FAC-main(9)
    idle: waiting for some task to do @ #x211d07a2>
   :TYPE :MAIN)))
(:SAVED-ANSWERS NIL)

=== TASKS
(:PROJECTS NIL)
(:WAITING-TASKS NIL)

=== APPEARANCE
(:WINDOW #<AGENT-WINDOW: SA_FAC>)
(:COM-WINDOW NIL)
```

Figure 2.3: Some details of the content of the factorial agent

2.1.2 Creating and Sending Messages

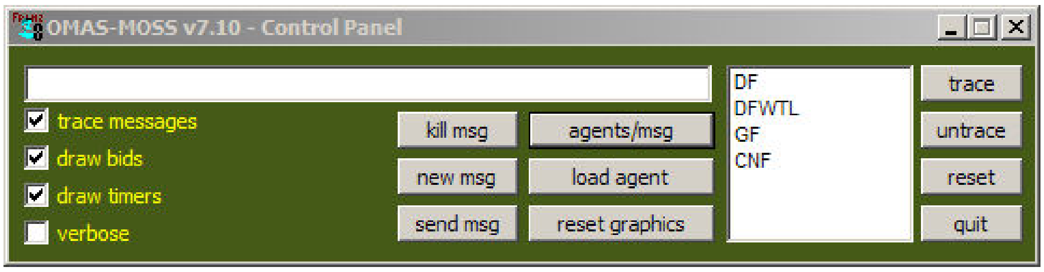
Viewing the List of Available Messages

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.2.4).

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

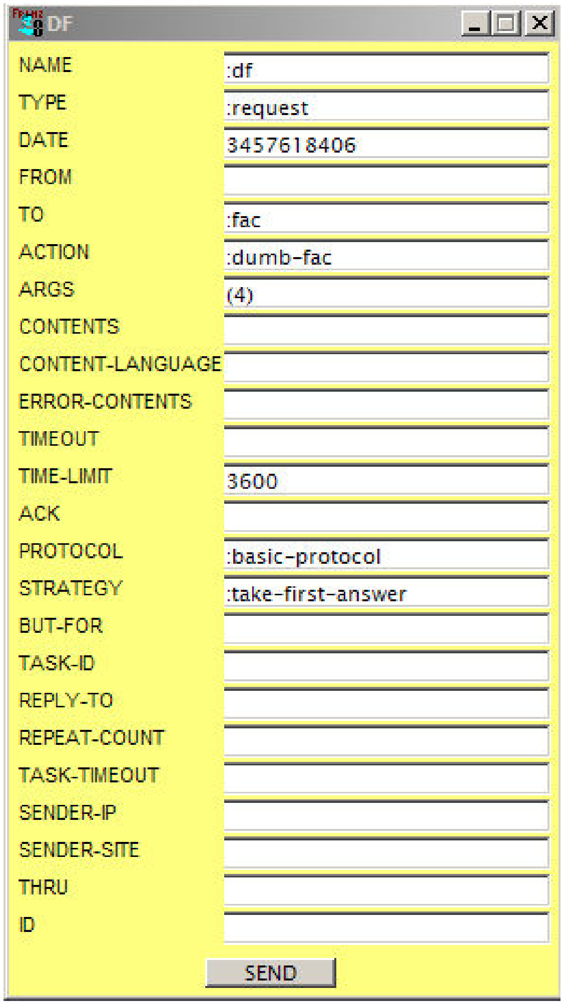
Examining a Message

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.2.5.



DF
DFWTL
GF
CNF

Figure 2.4: List of messages showing four predefined messages



NAME	:df
TYPE	:request
DATE	3457618406
FROM	
TO	:fac
ACTION	:dumb-fac
ARGS	(4)
CONTENTS	
CONTENT-LANGUAGE	
ERROR-CONTENTS	
TIMEOUT	
TIME-LIMIT	3600
ACK	
PROTOCOL	:basic-protocol
STRATEGY	:take-first-answer
BUT-FOR	
TASK-ID	
REPLY-TO	
REPEAT-COUNT	
TASK-TIMEOUT	
SENDER-IP	
SENDER-SITE	
THRU	
ID	

Figure 2.5: Message editing window showing the content of the DF (dumb factorial) message

The message shown Fig.2.5 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.2.5 is TYPE, TO, ACTION and ARGS, the rest is optional.

Sending a Message

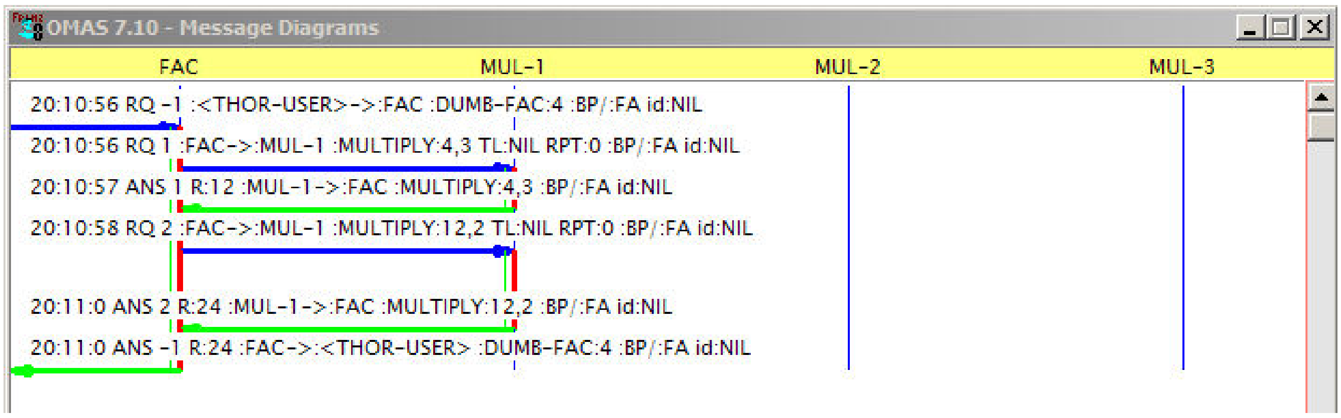
There are two ways of sending a message:

- clicking on the `send` button of a message editing window (Fig.2.5)
- selecting a message in the message list and clicking on the `send meg` button of the control panel.

2.1.3 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. 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 cllicking the send msg button, we see the exchanges of messages (Fig.2.6).



```
20:10:56 RQ -1 :<THOR-USER>->:FAC :DUMB-FAC:4 :BP/:FA id:NIL
20:10:56 RQ 1 :FAC->:MUL-1 :MULTIPLY:4,3 TL:NIL RPT:0 :BP/:FA id:NIL
20:10:57 ANS 1 R:12 :MUL-1->:FAC :MULTIPLY:4,3 :BP/:FA id:NIL
20:10:58 RQ 2 :FAC->:MUL-1 :MULTIPLY:12,2 TL:NIL RPT:0 :BP/:FA id:NIL
20:11:0 ANS 2 R:24 :MUL-1->:FAC :MULTIPLY:12,2 :BP/:FA id:NIL
20:11:0 ANS -1 R:24 :FAC->:<THOR-USER> :DUMB-FAC:4 :BP/:FA id:NIL
```

Figure 2.6: 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).

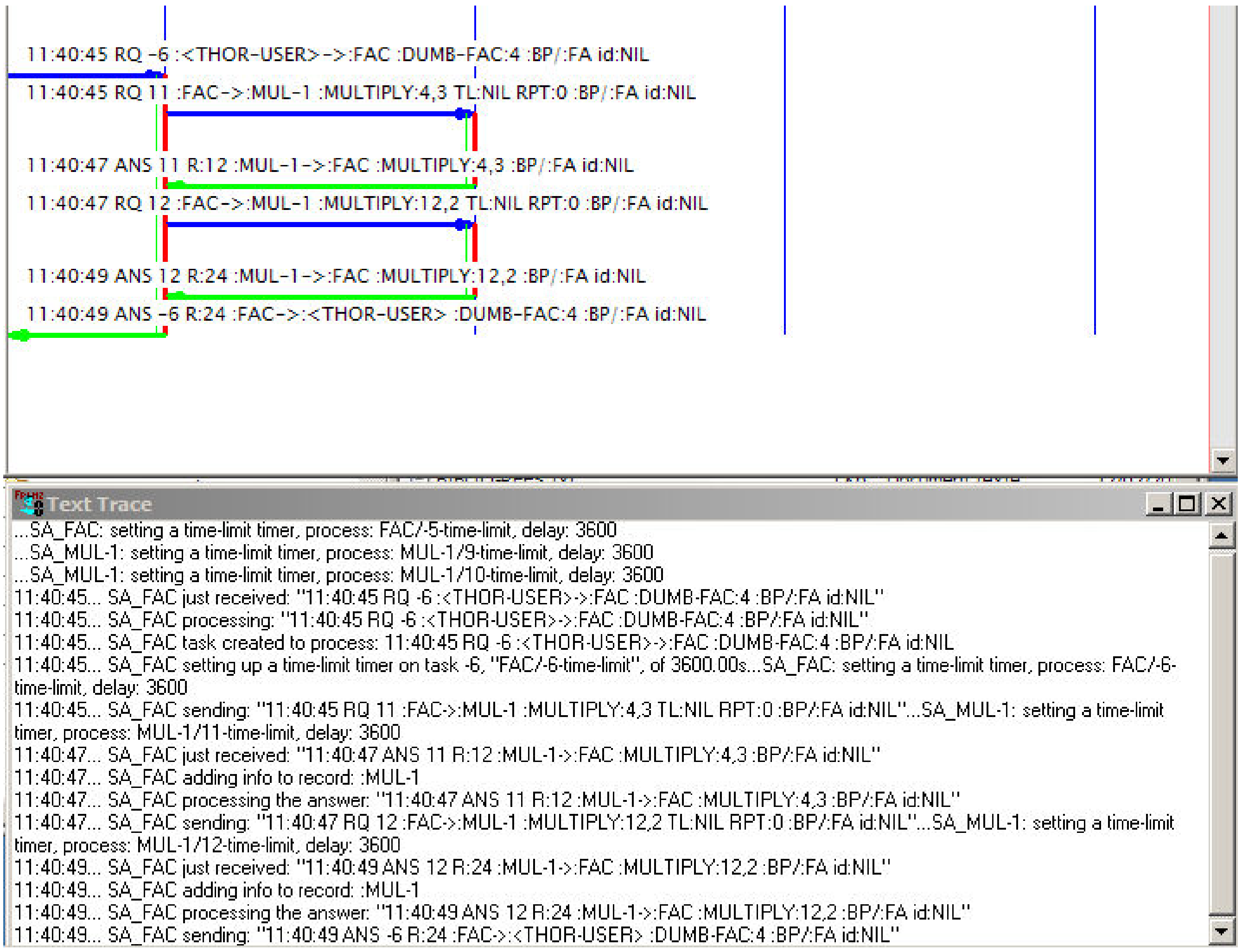
We can notice several things:

- 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[1]). Unchecking the `draw timers` check box in the control panel removes the timer lines.

2.1.4 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.2.7).



```
11:40:45 RQ -6 :<THOR-USER>->:FAC :DUMB-FAC:4 :BP/:FA id:NIL
11:40:45 RQ 11 :FAC->:MUL-1 :MULTIPLY:4,3 TL:NIL RPT:0 :BP/:FA id:NIL
11:40:47 ANS 11 R:12 :MUL-1->:FAC :MULTIPLY:4,3 :BP/:FA id:NIL
11:40:47 RQ 12 :FAC->:MUL-1 :MULTIPLY:12,2 TL:NIL RPT:0 :BP/:FA id:NIL
11:40:49 ANS 12 R:24 :MUL-1->:FAC :MULTIPLY:12,2 :BP/:FA id:NIL
11:40:49 ANS -6 R:24 :FAC->:<THOR-USER> :DUMB-FAC:4 :BP/:FA id:NIL
```

```
...SA_FAC: setting a time-limit timer, process: FAC/-5-time-limit, delay: 3600
...SA_MUL-1: setting a time-limit timer, process: MUL-1/9-time-limit, delay: 3600
...SA_MUL-1: setting a time-limit timer, process: MUL-1/10-time-limit, delay: 3600
11:40:45... SA_FAC just received: "11:40:45 RQ -6 :<THOR-USER>->:FAC :DUMB-FAC:4 :BP/:FA id:NIL"
11:40:45... SA_FAC processing: "11:40:45 RQ -6 :<THOR-USER>->:FAC :DUMB-FAC:4 :BP/:FA id:NIL"
11:40:45... SA_FAC task created to process: 11:40:45 RQ -6 :<THOR-USER>->:FAC :DUMB-FAC:4 :BP/:FA id:NIL
11:40:45... SA_FAC setting up a time-limit timer on task -6, "FAC/-6-time-limit", of 3600.00s...SA_FAC: setting a time-limit timer, process: FAC/-6-
time-limit, delay: 3600
11:40:45... SA_FAC sending: "11:40:45 RQ 11 :FAC->:MUL-1 :MULTIPLY:4,3 TL:NIL RPT:0 :BP/:FA id:NIL"...SA_MUL-1: setting a time-limit
timer, process: MUL-1/11-time-limit, delay: 3600
11:40:47... SA_FAC just received: "11:40:47 ANS 11 R:12 :MUL-1->:FAC :MULTIPLY:4,3 :BP/:FA id:NIL"
11:40:47... SA_FAC adding info to record: :MUL-1
11:40:47... SA_FAC processing the answer: "11:40:47 ANS 11 R:12 :MUL-1->:FAC :MULTIPLY:4,3 :BP/:FA id:NIL"
11:40:47... SA_FAC sending: "11:40:47 RQ 12 :FAC->:MUL-1 :MULTIPLY:12,2 TL:NIL RPT:0 :BP/:FA id:NIL"...SA_MUL-1: setting a time-limit
timer, process: MUL-1/12-time-limit, delay: 3600
11:40:49... SA_FAC just received: "11:40:49 ANS 12 R:24 :MUL-1->:FAC :MULTIPLY:12,2 :BP/:FA id:NIL"
11:40:49... SA_FAC adding info to record: :MUL-1
11:40:49... SA_FAC processing the answer: "11:40:49 ANS 12 R:24 :MUL-1->:FAC :MULTIPLY:12,2 :BP/:FA id:NIL"
11:40:49... SA_FAC sending: "11:40:49 ANS -6 R:24 :FAC->:<THOR-USER> :DUMB-FAC:4 :BP/:FA id:NIL"
```

Figure 2.7: 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
- it created a task

[1]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

- ... 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.

2.1.5 Miscellaneous Control Buttons

Control Panel

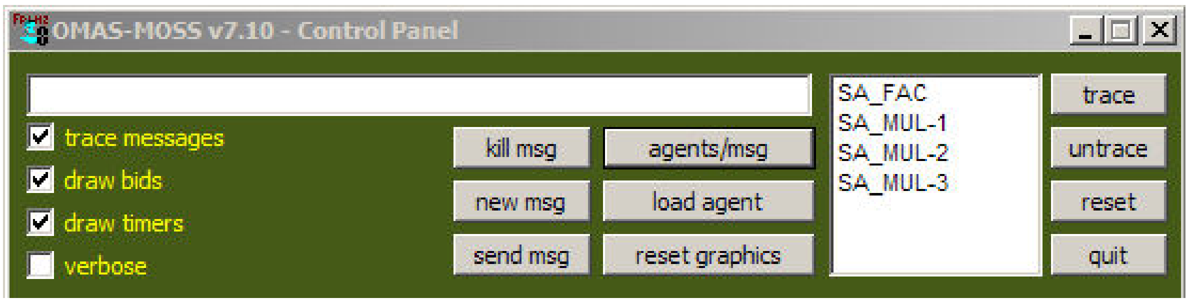


Figure 2.8: Agents from the factorial application

Let us summarize the role of the control buttons appearing 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 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

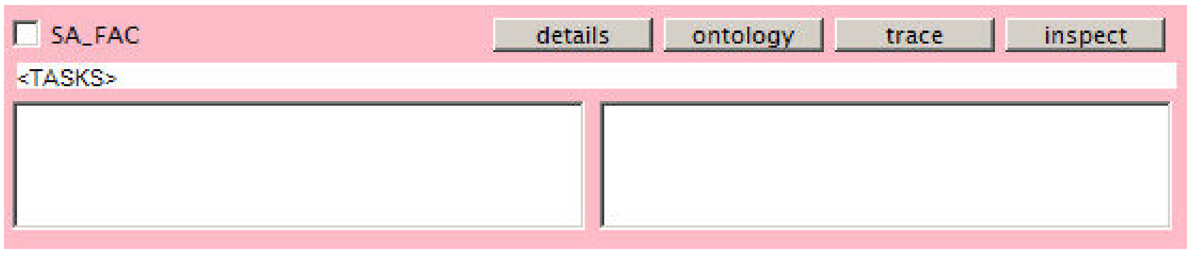


Figure 2.9: Agent window

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

- Top row:

  details: Opens a window giving some details on the content of the agent.

  ontology: Opens a window showing the content of the associated ontology (all agents do not have necessarily an ontology).

  trace: toggles tracing/untracing an agent (coupled with the verbose check box of the control panel).

inspect: opens a Lisp inspector for inspecting the agent structure (reserved for Lisp specialists).

- Second row:

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

Chapter 3

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 supported by OMAS.

3.1 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 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.3.1). A local coterie does not necessarily contain a personal 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 their own protocol and use TCP. When connected to a foreign platform the protocol 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.

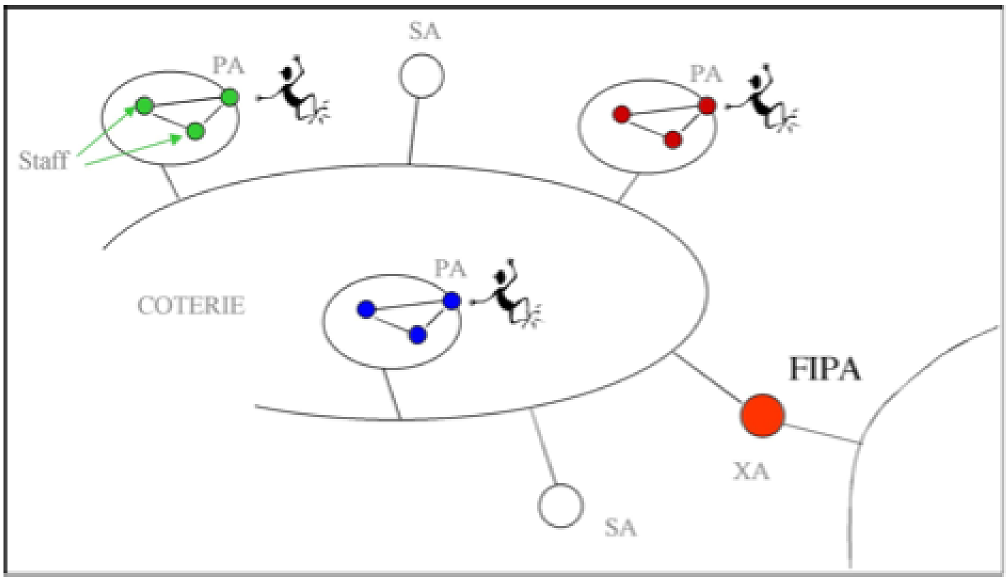


Figure 3.1: Coterie showing local coteries and inter-coterie link

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

3.2 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.

3.2.1 Service Agents

Service Agents are regular agents that provide a set of services. They can be thought as Web Services with the difference that they may not answer even if they can do the job. Their structure is shown Fig.3.2.

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.

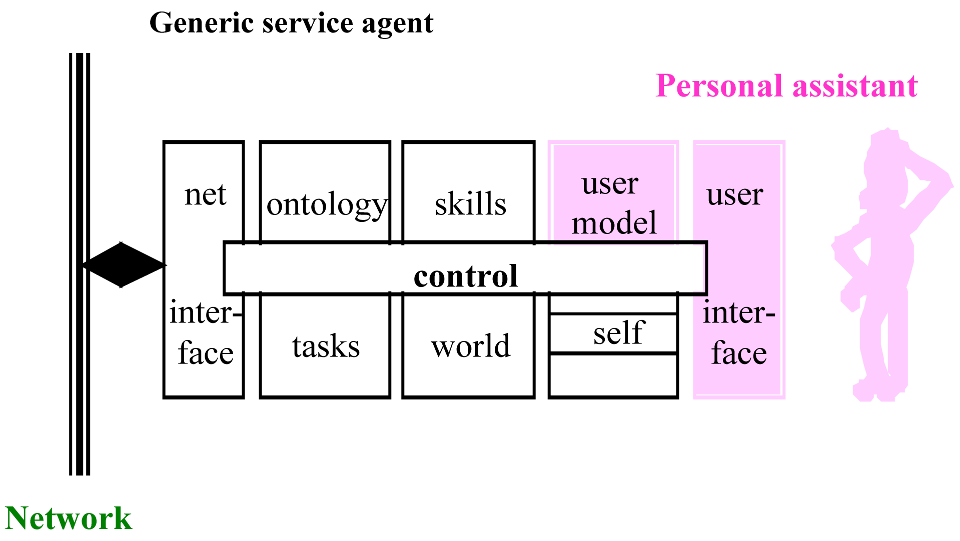


Figure 3.2: The structure of a Service Agent / Personal Assistant (with grey additions)

3.2.2 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.

3.2.3 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.

3.2.4 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 4

Service Agent

Contents

This chapter describes the structure of a Service Agent (SA), and various mechanism 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.

4.1 Structure

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

```
(defagent :OSCAR)
```

The macro can take several options:

- :learning
- :master
- :persistency
- :hide

Learning means that the agent keeps the results of performing tasks in a cache and if it is asked to do the same thing another time, it will read the result from its cache. It does not really implement any kind of learning, and can only be used in specific situations.

```
(defagent :OSCAR :learning t)
```

Master means that the agent is a Staff Agent serving a master (usually a Personal Assistant). The agent can only give answers to its master. However, it can send messages to any other agent on the platform.

```
(defagent :CALENDAR :master (:oscar))
```

Note. An agent may have several masters. This may be convenient when a user has several PAs.

Persistency means that the agent has a persistent store in which it keeps its ontology and knowledge base.

```
(defagent :OSCAR :persistency t)
```

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.

Hide means that the agent remains hidden from the user. E.g. the agent handling the graphics window is hidden from the user.

```
(defagent :demon :hide t)
```

This is normally reserved for system agents that are not displayed to the user.

4.2 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).

4.2.1 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 he 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:

- acknowledge-fcn
- bid-cost-fcn
- bid-quality-fcn
- bid-start-time-fcn
- dynamic-fcn
- how-long-fcn
- how-long-left-fcn
- preconditions
- time-limit-fcn
- timeout-handler

4.2.2 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 doen 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.

4.2.3 Bid Cost Option

This option allows the user to specify a function for computing the cost of doing a task. Far 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))
```

4.2.4 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.

4.2.5 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.

4.2.6 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 send 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. Events and dispatching is done by the platform. An agent could do several factorial computations in parallel, there is no danger of mixing the data.

4.2.7 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 do the job.

4.2.8 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.

4.2.9 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.

4.2.10 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.

4.2.11 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 allowed 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.

4.2.12 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

In a message 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)
```

The subtask will be aborted (default behavior if no timeout handler has been defined) after 2 seconds. By default the task itself we 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 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` 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*)
```

4.2.13 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 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.

4.3 Goals

4.3.1 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 set 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 the reactive agents).

4.3.2 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 is an object having 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 is 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.

4.3.3 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.

4.3.4 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.

4.3.5 Simple Tests

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
```

4.3.6 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.

4.4 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 saved data, in particular during the dialogs with a PA.

4.4.1 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)
```

4.4.2 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.

4.5 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.

4.5.1 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.

4.5.2 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.

4.6 Ontology

OMAS ontologies are formalized using the MOSS representation language. The approach is detailed in chapter **??**.

4.7 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 **??**.

4.8 Appendix A - Service Agent Structure

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

```
 an agent has the following structure, implementes as sub-objects
   name                  ; qualifies the agent (external name) e.g. SA_ADDRESS
   key                   ; keyword corresponding to its name
   master-if-staff       ; list of masters (keyword) for a staff service agent
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
   input-log             ; log of all input messages
   output-log            ; log of all input messages
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
world
   acquaintances         ; other known agents
   environment           ; (obsolete: replaced by data)
   external-services     ; knowledge of other agents
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
   processs-list         ; list of processes except for scan, min, and assistant
   pending-tasks         ; subtask list needed when executing the skill
                         ; sould be part of the corresponding task
   pending-bids          ; active bids still pending
                         ; used by the CNet protocol
   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)
   to-save                   ; contains a list of objects to be saved during the
                             ; next transaction
   moss-version-graph        ; version (configuration) graph
   package                   ; agent package (for ontology symbols)
tasks
   projects                  ; description of complex tasks (unused yet)
   waiting-tasks             ; description of task being processed (learning agents)
appearance
   h-pos                     ; horizontal position for graphics display
   v-pos                     ; vertical position for graphics display
   window                    ; output window
   comm-window               ; window for communicating with the user
   window-position           ; output window position
   thread                    ; (?)
skills
   skills                    ; list of functions to be applied
   rule-sets                 ; list of rule sets to be applied
master-bins (implemented in the assistant file)
   waiting-messages          ; messages to be processed by the master
   pending-requests          ; issued requests waiting for answer
   discarded-messages        ; messages discarded by the assistant (scanner)
   waiting-answers           ; answers to be examined
   to-do                     ; input string from the master
   conversation              ; current conversation
   conversation-state        ; unused
   conversation-context      ; unused
   dialog-header             ; start of the initial dialog
   pending-task-id           ; unused?
   answer                    ; answer to a send-request message
transfer (implemented in the POSTMAN file)
   destination               ; destination ip
   port                      ; transfer port
   transfer-protocol         ; tranfer protocol (default TCP)

process                      ; process running the agent
mbox-process                 ; process attached to the input-messages mailbox
traced                       ; t if agent must be traced, nil otherwise
```

4.9 Appendix B - Agent Skill Structure

```
skills are CLOS objects
```

Chapter 5

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 the 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 a natural language dialog, through keyboard input or vocal input, in French or in English. Specific mechanisms have been developed to allow constructing such dialogs easily.

5.1 Creating a Personal Assistant Agent (PA)

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

```
CG-USER(12): (defassistant :zoe)
#<AGENT ZOE>
```

However, if the agent ZOE has indeed been created, there is no visible means of interaction other than sending it messages like to a standard service agent. The main reason is the necessity to set up a specific interaction machinery that consists of a set of tasks, an ontology, a dialog and an interaction window. This is normally done by creating specific files containing the required elements. Such files are then loaded automatically when loading an application (See Chapter 1 §1.2.3).

When loading an agent file, a default task file and a default dialog file, an interaction window appears as shown Fig.??. One can then communicate with the PA by typing things in the master's pane of the window.

5.2 PA Interaction Window

5.2.1 Basic Mechanism

The Master (user) communicates with her personal assistant through the interface (Fig.??). Communication is usually done as a dialog carried in the right hand side of the interface, using the Assistant and the Master areas.

The left hand side of the interface is devoted to handling communications with external agents, corresponding to messages sent or received. It includes four areas:

- **Answers/Info**: an area containing answers to requests done by the PA on behalf of the master.
- **Tasks to do**: the list of requests that are asked to the master and that the agent could not process by itself.
- **Pending requests**: request 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.

5.2.2 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 yes the message is kept, otherwise the message is discarded.

5.2.3 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.

5.2.4 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.

5.2.5 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.

5.3 Principle of the Dialog

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.

Two 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 it has not been published yet.

Tasks are described in Section 5.4 and dialogs are described in Section dialogs.

5.4 Tasks

5.4.1 The Library of Tasks

Whenever the user asks something, its 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 concept of task. 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*>, <*message action*> indicates which staff agent and what skill are concerned with the input, and is used when no dialog is feasible, which is the case considered in this paper.

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

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

5.5 Dialogs

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

- a general high 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.

5.5.1 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. The details of the algorithm are given in the following document:

```
UTC/GI/DI/N217 - OMAS v7 - Personal Assistant (June 2008)
```

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 too much worrying about the mechanism.

We examine first the top-level conversation (dialog), then an example of a sub-conversation (sub-dialog).

5.5.2 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.5.1. 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.

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

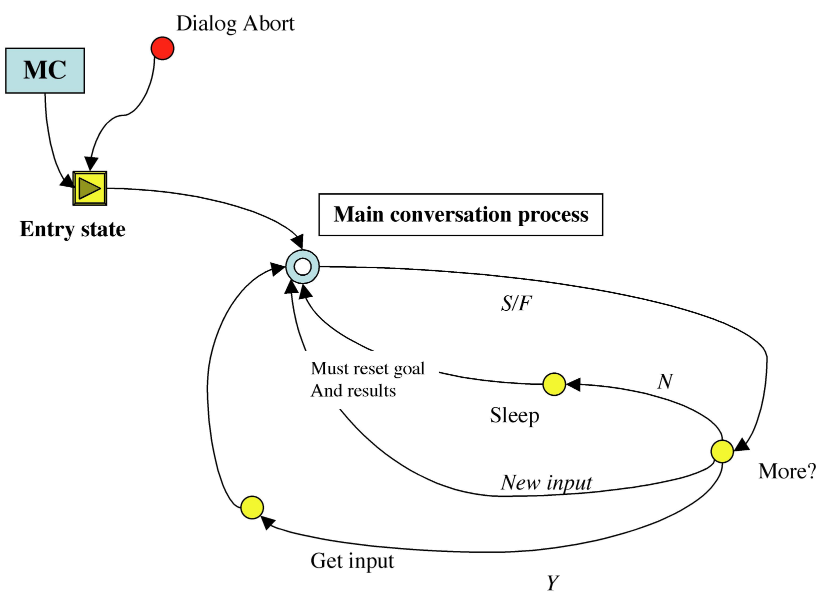


Figure 5.1: PA Main Conversation, showing the "Main Conversation Process" sub-dialog

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 compile eval)
  (proclaim '(special
              _MAIN-CONVERSATION
              ...
              )))

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

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

```
(defindividual
  Q-DIALOG-HEADER
  (HAS-LABEL "Main conversation loop")
  (HAS-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 load eval)
  (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))
  )
```

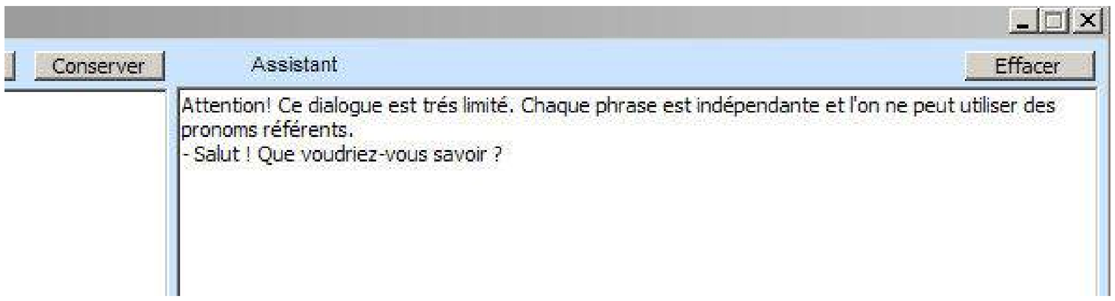
It reads as follows:

- 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.5.2):

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



Attention! Ce dialogue est trés limité. Chaque phrase est indépendante et l'on ne peut utiliser des pronoms référents.
- Salut ! Que voudriez-vous savoir ?

Figure 5.2: Initial opening of the Main Conversation 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.5.3.

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
```

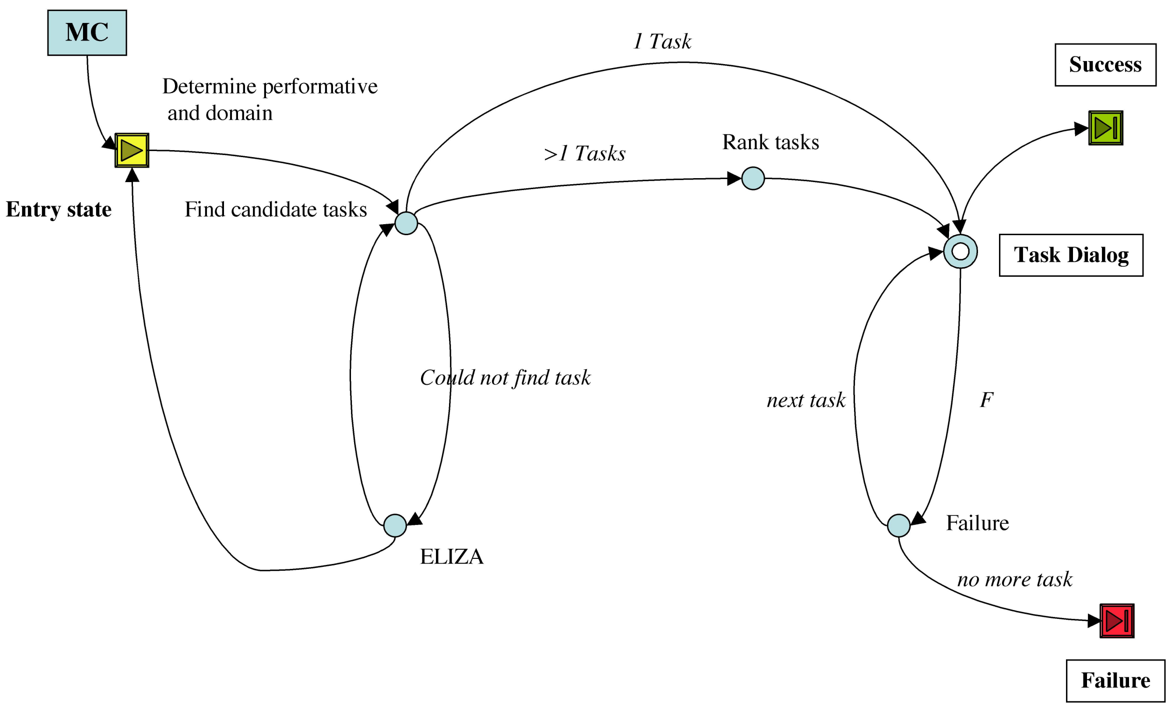


Figure 5.3: Main Conversation Process sub-dialog

```
(eval-when (compile load eval)
  (proclaim '(special
              _mc-eliza
              _mc-failure
              _mc-find-performative
              _mc-find-task
              _mc-select-task
              _mc-task-dialog)))
```

This declares the names of the various states (nodes of the automaton). In addition, the sub-dialog is declared through the defsubdialog macro as follows:

```
;;;----------------------------------------------------- (MC) PROCESS-CONVERSATION

(defsubdialog
  _process-conversation
  (:label "Process conversation")
  (:explanation
   "Processing steps of the main conversation.")
  )
```

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" (?* ?x))
            ("quel" "est" (?* ?x))
            ("quelle" "est" (?* ?x))
            ("quels" "sont" (?* ?x))
            ("quelles" "sont" (?* ?x))
            ("qu" "est-ce" "que" (?* ?x))
            ("quoi" (?* ?x))
            ("quand" (?* ?x))
            ("où" (?* ?x))
            ("pourquoi" (?* ?x))
            ("qui" (?* ?x))
            ("combien" (?* ?x))
            ("comment" (?* ?x))
            ("est-ce" "que" (?* ?x))
            ("est-il" (?* ?x))
            ("est" "il" (?* ?x))    ; interface vocal ?
            ("est-elle" (?* ?x))
            ("est" "elle" (?* ?x))
            ("sont-ils" (?* ?x))
            ("sont" "ils" (?* ?x))
            ("sont-elles" (?* ?x))
            ("sont" "elles" (?* ?x))
            ("a-t-il" (?* ?x))
            ("a-t-elle" (?* ?x))
            ("ont-ils" (?* ?x))
            ("ont-elles" (?* ?x))
            ((?* ?x) "c" "est" "quoi" (?* ?y))
            ((?* ?x) "c" "est" "qui" (?* ?y))
            ((?* ?x) "?")
            )
           :set-performative (list :request)
           :target _mc-find-task)
 (:patterns (((?* ?x) "notez" (?* ?y))
            ((?* ?x) "noter" (?* ?y)))
           :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 context 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 expressions 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 5.7.1).

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 /input: ~S, package: ~S" input *package*)
    ;; 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))
    ;; 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-TASK conversation) task-list)
      `(:transition ,_mc-task-dialog))
     (t
      ;; otherwise must select one from the results
      (setf (HAS-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-TASK state-context) task-list)
      `(:transition ,_mc-task-dialog))
     (t
      ;; otherwise must select one from the results
      (setf (HAS-TASK-LIST state-context) 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-TASK-LIST slot
  (let* ((task-list (HAS-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)
    ;; 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-TASK-LIST conversation) task-list)
    (setf (HAS-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-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 we remove the scores from the list, select the first task and remove it from the list:

```
;; remove the weights
(setq task-list (mapcar #'car pair-list))
;; select the first task of the list
(setq selected-task (pop task-list))
```

Then we update the context object and make a transition to the "Task Dialog" state:

```
;; save the popped list in the task-list slot of the conversation object
(setf (HAS-TASK-LIST conversation) task-list)
(setf (HAS-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 5.6. 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)
  )

;;;********** must check for possible problems
(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-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-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-TASK-LIST task-list)
        ;; add it to conversation
        (send conversation '=replace 'HAS-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.

5.6 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. The rationale is to control the order in which required missing parameters are asked from the master.

5.6.1 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?


(eval-when (compile load eval)
  (proclaim '(special
              _ph-print-global-help
              )))
```

The corresponding automaton is shown Fig.5.4.

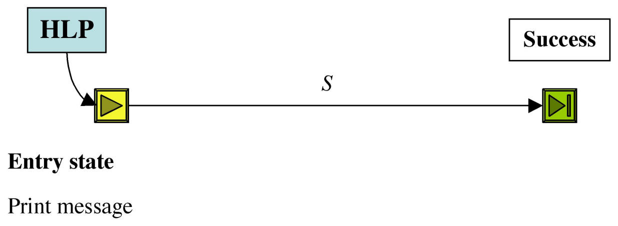


Figure 5.4: Print Help sub-dialog

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")
  (: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 albert::pa_albert))
      )
     (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))
  )
```

5.6.2 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.

(eval-when (compile load eval)
  (proclaim '(special
             _get-address-dialog  ; required by defstate
             _gad-entry-state
             _gad-dont-understand
             _gad-try-again
             _gad-sorry)))
```

This corresponds to the automaton shown Fig.5.5.

Get Address Entry State

The code is as follows:

```
;;;----------------------------------------------- (GAD) GET-ADDRESS-CONVERSATION

;;; The simplest use is to take text in the HAS-DATA slot of the Q-CONTEXT
;;; and send a message to :SA-ADDRESS
```

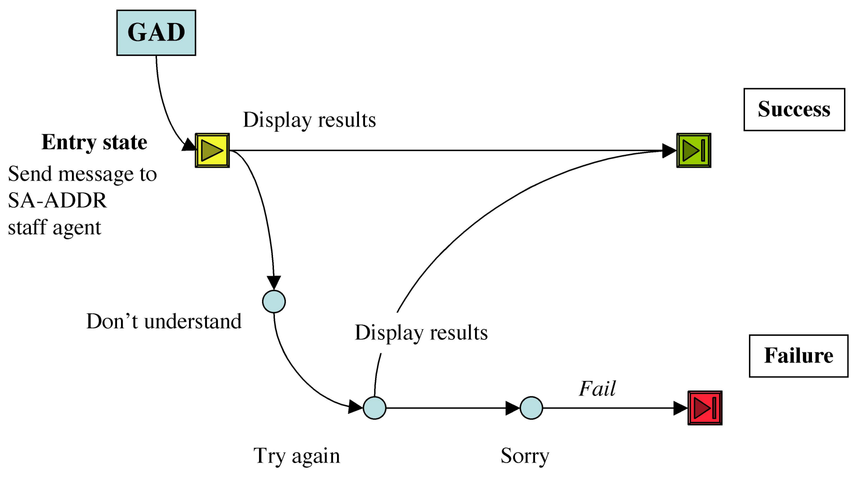


Figure 5.5: Get Address sub-dialog

```
(defsubdialog
  _get-address-conversation
  (:label "Get address conversation")
  (:explanation
   "Master is trying to obtain an address.")
  )
;;;------------------------------------------------ (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-address :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::get-data moss::conversation))
   (:language . :fr)))
```

Data includes the input from the user, and language specifies the language to be used because staff agents can have multilingual capacities.

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 commented out and should be rewritten.

5.7 System Internals

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

5.7.1 Viewing the defstate Code

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

- set the moss::*debug* global variable to T;
- 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
```

5.7.2 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.

5.7.3 Tracing the Dialog

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

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

5.8 More on the **defstate** Macro

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

5.8.1 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 (((?* ?x) "définir" (?* ?y))
               ((?* ?x) "définition" (?* ?y))
               ((?* ?x) "expliquer" (?* ?y))
               ((?* ?x) "expliquez" (?* ?y))
               ((?* ?x) "que" "veut" "dire" (?* ?y))
               ("c" "est" "quoi" (?* ?y) "?")
               ((?* ?y) "c" "est" "quoi" (?* ?x))
               )
              :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))
  )
```

5.8.2 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))
  )
```

5.8.3 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))
  )
```

5.8.4 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 has been demonstrated in the

Main Conversation dialog.

5.9 Some Problems

5.9.1 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.

5.9.2 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.

5.10 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 5.1.

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

5.10.1 Global Syntax

The global syntax is:

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

where

Table 5.1: Structure of the conversation object

Property	att/rel	role
OUTPUT-WINDOW	att	panel in which to write questions
INPUT-WINDOW	att	panel from which to read master's data
STATE Q-STATE	rel	current-state of the conversation
INITIAL-STATE Q-STATE	rel	initial state
AGENT	att	PA ID implied in the conversation
DIALOG-HEADER	rel	main conversation header
SUB-DIALOG-HEADER	rel	current sub-conversation
FRAME-LIST	att	list of return addresses of sub-dialogs
TEXT-LOG	rel	record conversation texts
TASK-LIST	rel	list of potential tasks for dialog
TASK	rel	current task
FACTS	att	FACT base structured as an alternated list

- 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);
- 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 5.2.

5.10.2 Global Options

They are:

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

5.10.3 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:

Table 5.2: 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>) |
		(:eliza) |
		(:execute-preconditions <Lisp code>) |
		(:question <string>*) |
		(:question-no-erase <string>*) |
		(:reset) |
		(: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>) |
		(:on-failure <action clause> ∗ <transition>) |

- :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 grom 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.
- :reset, reset the conversation by restarting it at the beginning. Cleans FACTS and transfers to the entry state.
- :send-message args, send 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, send 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}, print text e.g. prior to asking a question.
- :wait, set OMAS to wait for master's input. Presumably the question has been asked in a previous state.

5.10.4 Options for the =resume Method

- :answer-analysis, indicates that a special method named =answer-analysis will be applied to the data (INPUT). The method must return a transition state or the :abort keyword, in which case the conversation is locally restarted.
- :execute, executes the corresponding Lisp code.
- :otherwise, always fires (should be last clause)
- :transitions, specifies the transitions after doing some tests (details in the next paragraph).

The :transitions option is the important one since it contains the transition rules. the syntax of its arguments is the following:

- :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 is ("*failure*")
- :patterns ...,
  - :keep,
  -
- :otherwise ..., always fire (should be the last option).
- :print-answer print-function, prints the content of the FACTS/answer area applying the print-function function.
- :reset

- :save
- :self method args
- :sub-dialog dialog-header
- :target state
- :test expr

5.11 PA Communications

5.12 Creating a Foreign Personal Assistant

Chapter 6

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.

6.1 Introduction

6.1.1 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) 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.

6.1.2 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) or on different sites (remote coterie).

All messages received by the postman will be 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;
- messages addressed to the local user to be posted into the coterie control panel.

6.1.3 Protocol

By default the communication protocol is TCP/IP. Thus, each agent is both a server and a client. Messages use sockets on port 52008.

6.1.4 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 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, which is done by a connect skill.

Receiving

When a postman is created, it starts a specific receiving process that will wait for a connection on port 52008. 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.

6.1.5 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. One must take care of possible looping conditions.
- IPs: Machines in different research center 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 OMAS server). 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.

6.2 Implementation

6.2.1 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 pairs <postman-name, IP>. Usually, IP is the external IP through which messages can be routed. This list is first initialized when the postman is created, but may be extended if messages are received from unknown 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.
- **local site**. A key specifying the local site, e.g. :UTC or :CIT.
- **site-ip**. The IP of the postman site (local coterie).

- **send-socket**. The socket object for sending messages.
- **receive-socket**. The socket for receiving external messages.
- **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.

6.2.2 Creating a Postman

When creating a postman, one must specify a number of parameters:

- the name of the postman as a key, e.g. :UTC; if the postman is a server for the site, this will also be the site name;
- the machine IP if known and fixed (cannot be done for machines connected in DHCP mode);
- the list of known postmen as a list of pairs <postman key, IP>, where the IP may be external or internal;
- the fact that the postman is a server (:server t);
- an optional user-defined receiving function (:receiving-fcn <user-defined function>).

6.2.3 Connecting a New Remote or Local Coterie

When one sends a connect message to the postman directly or using the postman window, the postman tries to open a socket onto the remote site or local target coterie. If it succeeds, then the remote postman name and IP are added to the list of connected postmen. If it fails, e.g. on a timeout, then they are not added to the list. When the connection succeeds, the agent uses the socket to retrieve its local IP address (only if it is unknown and on the first time).

6.2.4 Sending a Message

Sending a message is done by the send skill of the postman as defined by the user. However, it is possible to use the OMAS postman-send function that does the following:

1. It computes the list of postmen to which the message is to be sent by removing from the active list 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. For each send message, it created a new thread to avoid delaying other messages in case of a socket error, calling the send-remote function. The rest is done by the send-remote function.
5. The target postman is checked to see if it is still active (not very useful).
6. The message to send is converted into a coded string.
7. A socket is created for connecting to server (target postman).
8. If an error is detected, the target postman is removed from the list of active postmen and the local postman window is updated.
9. Otherwise the message is sent and the socket is closed.

6.2.5 Receiving a Message

When the postman was create a special thread with the postman-receiving function has been set up. The function created a passive socket, then enters a loop creating a stream and connection socket. It then waits for incoming messages on the receiving socket. When a message comes in the following happens:

1. The message (string) is read and the connection socket is closed.
2. If the message is empty, go wait from the next one.
3. The message is converted into an OMAS message object (demarshalling). If this fails we go wait for the next message.
4. 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.
5. Otherwise, the id is added to the list of received message ids.
6. The name and IP of the sending postman are extracted from the message.
7. If the sending postman is known and already active, nothing is done.
8. If the sending postman is known but not active, it is added to the list of active postmen.
9. If the sending postman is not known, then an entry is added to the list of active postmen.
10. Then, the message is put on the loop of the local coterie. Doing so results in the postman receiving the message again. The send skill is recalled and the postman will transfer the message to the active sited that were not yet visited by the message.

6.2.6 Disconnecting a Coterie

The :disconnect skill simply remove the postman info from the list of active postmen.

6.2.7 Resetting Connections

In case of problems the :reset skill disconnects all active postmen, closes all sockets, kills the receiving process and creates a fresh one. It updates the postman window if opened.

6.2.8 Getting the Connection Status

The :status skill print info about the status of the connection into the console.

6.3 Example of Minimal Postman

6.3.1 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 postman-send function provided by OMAS.

```
;;;-*- 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.
;;; This uses the new v.7.16 implementation of the postmen

#|
2010
 0501 Re-creation
|#

;;;===============================================================================
;;;
;;;                      Creating postman
;;;
;;;===============================================================================

(omas::defpostman :UTC
             :redefine t
             :server t
             :known-postmen ((:CIT . "219.166.183.59")
                             (:TECPAR . "200.183.132.15"))
             )

;;;===============================================================================
;;;
;;;                                     skills
;;;
;;;===============================================================================

;;; ================================ skill section ========================= SEND

(defskill :SEND :UTC
  :static-fcn static-SEND
  )

(defun static-send (agent in-message message-string message)
  "skill that sends every message seen by the postman to active remote sites,
   with the exception of the user and system messages.
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))
  ;; we transfer messages only if some platforms are connected and active
  (omas::postman-send agent message)
  (static-exit agent :done))

;;; ===========================================================================

:EOF
```

6.3.2 User Point of View

The user must provide the Postman code. When the postman file is loaded, a receiving process is created waiting for messages on port 52008. This can be checked by examining the list of active processes.

To connect a remote coterie one can use the postman window (Fig.6.1), select the target remote postman and click the connect button.

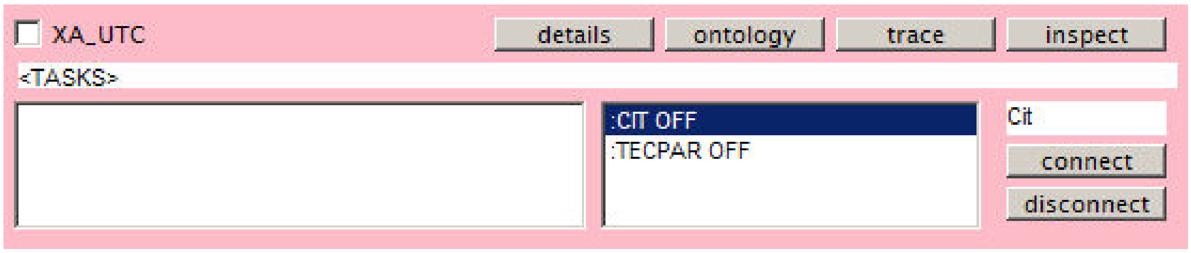


Figure 6.1: Postman Window

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.

Chapter 7

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.

7.1 Example

7.1.1 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.

7.1.2 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.

7.1.3 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.

7.1.4 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)
```

7.2 Improvements

Several types of improvements can be done.

7.2.1 Better Rule Format

Since rules are translated internally into Lisp structure, any format keeping the semantics of the rules could be defined.

7.2.2 Other types of Inferences

Forward Chaining is currently implemented. One could easily implement backward chaining, or mixed strategies.

7.2.3 Order of the Rule System

The exemple 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.

7.2.4 External Inference engines

Exteral inference engines can be called, using the foreign function mechanism.

7.2.5 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).

7.3 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 8

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.

8.1 Agent Communication Language

8.1.1 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 8.1 and described in the subsequent sections.

Table 8.1: 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

8.1.2 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.

8.1.3 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.

8.1.4 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.

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 8.1.9 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.

8.1.5 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, expecially 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.

8.1.6 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.

OMAS/ACL The content language is an associted 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.
- data-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.
- answer-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.

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.

8.1.7 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 OMAS/ACL Used by the contract-net protocol to allocate tasks and inform loosers with a single cancel-grant message. Addressing the message could be done with a predicate in the receiver parameter, however, this is a shorthand syntax specific to the cansel-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 initiatior of the protocol must assign a non null value to thr 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 wich 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 nex 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 y the agent implementor.

OMAS/ACL The parameter is time-limit. The time howeer specifies the maximum delay for the execution of the subcontracted task on the receiver's platform. The rational 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 two values: take-first-answer or collect-answers. In the latter case a specific timeot 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. To be removed.

8.1.8 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.

8.1.9 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.

8.2 OMAS Content Language

OMAS does not impose any particular content language. However, a simple structured language can be used for simple applications.

8.2.1 Overall Approach

The result of working on several application 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.

[1]Please refer to the MOSS documentation for the specification of a MOSS query.

8.2.2 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.

8.3 Network Interface

8.3.1 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 environment 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 ow the different protocols are implemented.

8.3.2 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

8.3.3 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 the `omas::*net-broadcast*` 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 "
"-5;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 seting 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.

8.3.4 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 512.

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 8.2.

Table 8.2: 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::ERROR-CONTENTS
10	OMAS::TIMEOUT
11	OMAS::TIME-LIMIT
12	OMAS::ACK
13	OMAS::PROTOCOL
14	OMAS::STRATEGY
15	OMAS::BUT-FOR
16	OMAS::TASK-ID
17	OMAS::REPLY-TO
18	OMAS::REPEAT-COUNT
19	OMAS::TASK-TIMEOUT

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 `*kTransferBufferSize*`. In our case its value is 4096.

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.

8.3.5 OMAS Net Interface

The interface between OMAS and the net (within a coterie) is controlled by global variables 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)
- MCL uses BSD sockets of Darwin, using the Apple CoreFoundation library (MacOSX)

Chapter 9

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.

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).

9.1 Convention

The user 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.

9.1.1 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.

9.1.2 Package

API functions are exported from the MOSS 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.

9.1.3 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).

9.1.4 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.

9.2 Global Variables

Starting with version 7.9, global parameters have been grouped into and instance of a CLOS object called SITE. The instance is called *omas* and is a global variable. All values can be accessed using the accessor functions and set be doing a setf on the accessor functions. Previously defined global variables are deprecated.

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.

9.2.1 Global Parameters

Table 9.1 contains a few parameters governing the state of the OMAS local coterie.

Table 9.1: 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)

9.2.2 Agents

Table 9.2 contains parameters related to agents.

Table 9.2: 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
site-name	"*unknown*"	name of the site of the local coterie
traced-agents	nil	list of agents that are traced

9.2.3 Messages

Table 9.3 contains parameters related to messages.

Table 9.3: 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

9.2.4 Graphics

Parameters described in Table 9.4 apply to the graphics trace display.

Table 9.4: 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

9.2.5 Programming Control

Table 9.5 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 9.5: Global parameters for programming control

Accessor	Default	Meaning
basic-processes	mp:*all-processes*	Microsoft: if true timeouts are displayed
	active-processes	MCL: id.

Continued on next page

Accessor	Default	Meaning
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)

9.2.6 Timings

Table 9.6 contains parameters related to default timings.

Table 9.6: 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

9.2.7 Tracing

Table 9.7 contains parameters governing tracing options.

Table 9.7: Global parameters for tracing

Accessor	Default	Meaning
omas-verbose	nil	toggles the debugging trace
traced-agents	nil	for printing conversations (MCL onlly)

Continued on next page

Accessor	Default	Meaning
trace-messages	t	if true print message text in graphics window

9.2.8 Network Interface

Table 9.8 contains parameters for handling local loop exchanges. Connections with remote coteries, or other systems are handled by transfer agents.

Table 9.8: 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

9.3 Functions

In this section, API functions have been grouped by features.

9.3.1 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.
```

9.3.2 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.
```

9.3.3 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.
;;;
```

9.3.4 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
;;;
```

9.3.5 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.
```

9.3.6 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.
;;;
```

9.3.7 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.
;;;
```

9.3.8 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
```

9.4 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 10

Persistency

Contents

10.1 Introduction

The OMAS (Open Multi-Agent System) platform has been developed over many years to let application designers build 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.

10.2 Overall Approach

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.

10.2.1 Agent Features

In addition to the MOSS requirements for each particular space (see N252L-MOSS-8-Persistency), agents have specific requirements related to the way they are loaded. Indeed, an agent is loaded from several files:

- the agent file defining the agent and containing skills and goals
- the agent ontology and knowledge base file
- the task file (PAs)
- the dialog file (PAs)

As a consequence, an agent environment contains a number of objects, namely:

- agents (self and others)
- ontology and KB objects
- objects describing skills
- PA conversations (a MOSS object)
- PA conversation states

- PA task objects

Adding persistency to the agent world raises some problems:

Some objects are created by OMAS while the agent is loaded, e.g. the object representing the agent (an instance of omas-object), the object describing the skills (instances of omas-skills), conversation and conversation states. The question is: should we treat such objects as we treat system objects? Another point concerns the ontology. Currently an agent ontology is defined in the AGENT-ONTOLOGY file. Should we save it onto disk? If we do so, then we loose all the comments contained in the ontology file. In addition all changes will occur through the editor, which is OK for small changes, but not for volume changes.

Another problem with some objects is that they are linked to KB objects. E.g. the object describing ALBERT is linked an individual person, the owner of the PA. Thus, if we store persons in the KB and recreate agent objects while loading the agent file, we end up with an inconsistency. The newly recreated ALBERT object will not have a link onto the person individual and the person individual may have a link onto a no longer existing object or, worse, onto an object different from the object linked in the past. A possible solution would be to list all possible conflicts and resolve such conflicts in a reconciliation phase after an agent is reloaded.

In practice, the problem is more difficult for a PA. Service agents tend to be simpler. Thus, we'd like to store KB objects and do not mind if the others are recreated when the agent is loaded. Anyway, if we want to save only the otology and related KB, we should have a more sophisticated stripping function when saving and restoring entry-points, or reloading agents and skills.

Editing objects conveniently: an important point to consider when saving objects into an object base is that we loose the commented representation we have in the original text files (comments are not kept after an object has been loaded into core). Hence, the question is: do we actually want to keep the task and dialog files or do we store the object into the object base. In the latter case it might be more difficult to debug the corresponding dialogs.

10.2.2 Programmer's View Point

From the programmer's point of view, we want the persistency mechanism to be relatively transparent. I.e., if an agent has persistent features that apply to its ontology and knowledge base, this must be transparent to the programmer. The programmer should safely assume that once the agent has been loaded, all objects are available in memory. However, if objects or concepts are modified, then the programmer must let the user decide whether they must be saved or not.

10.3 Implementation Version 0

10.3.1 Overall Approach

Several implementations are possible. We shall start with a simple one. The trade off is between what we want to save in the database and what we want to keep in text files. In practice we have two cases:

- when creating an application, it is easier to work with text files, because we keep updating the skills, goals. However, if we have a large knowledge base, it is better to keep it on disk.
- when the application is well defined and running, it is more convenient to store everything in the database, since there will be few modifications to the behavior of the agent.

This leads to the following assumptions:

1. A PA or an XA agent is not persistent; its ontology and KB are recreated each time the PA or the XA is restarted.

2. Only SAs may have a persistent object base, containing ontology and KB.
3. Skill objects (classes: omas-skill, counter, instances, entry-points, attached methods) are created anew each time one reloads an agent. They are not saved, thus should not be persistently modified by the application.
4. Ontology concepts, KB objects and related functions (methods) are saved into the object base.
5. When the agent is created the first time around, the KB files are created and the ontology and associated KB objects are saved in a bulk mode, with the exception of objects related to ontologies.
6. All new concepts and KB instances are created by program, i.e. they are not defined in the files being loaded (like AGENT-ONTOLOGY.lisp) or created or modified through the MOSS editor.
7. Objects are loaded as needed[1] by the QUERY mechanism, or by some kernel methods[2].

This has the following consequences:

1. entry-points should be stripped of the system items (including omas-skills).
2. After reloading the system, system entry points should be augmented with application data.
3. Saving a method should save the corresponding function.
4. Loading a method should load the corresponding function.
5. Saving a property should save the generic property.
6. Loading a property should load the generic property.

10.3.2 Agent Specifics and Relation to Persistent MOSS Ontologies

MOSS has already a mechanism to keep persistent ontologies and knowledge bases. The mechanism can be used for agents. However, an agent has additional objects in addition to ontologies, namely:

- the concept of agent and the individual representing itself
- the concept of skills and the individual skills
- the concept of goal and the individual goals

The concepts of agent, skill, and goal are independent of the changes that can occur when modifying an agent. Hence, the corresponding classes and properties can be saved onto disk. However, instances may change whenever the agent is modified. Since it is easier to modify an agent in a text file than in the object editor, we do not save the instances of agent, skill and goal, but recreate them each time an agent id reloaded. Consequently, the counters of the agent, skill and goal classes must be reset when saved into the database.

In addition, during the lifetime of the agent information could be attached to the instance of agent or to the instances of skills or goals. We assume that this is not the case, i.e. the instance of agent and the instances of skills and goals do not change during the lifetime of an agent. Note that the lifetime of an agent is limited to a session[3]. The major consequence is that we rebuild the instances whenever we reload the agent. Now, if we want to describe an acquaintance or an external skill or goal we could

[1]When they are not in core.
[2]In practice by service functions called by the kernel methods.
[3]A sesion can last several months.

use specific concepts like EXTERNAL-AGENT, EXTERNAL-SKILL or EXTERNAL-GOAL, which is not done in this implementation.

Summarizing, we have two cases: (i) "non persistent" agents, among which PAs, XAs, and some SAs, and "persistent" agents, saving ontology and knowledge base. We detail both cases.

10.3.3 Non-Persistent Agents

Such agents do not save anything on disk. However, they must initialize their name space with the relevant concepts, connected to the MOSS representation system. This uses the MOSS ontology mechanism, extending it to include Agent, Skill, and Goal concepts. Several cases must be examined: (i) loading an agent; (ii) leaving the system.

Loading a Non-Persistent Agent

An agent defines its ontology in its own namespace. In addition to the concepts of its ontology defined in the relevant files (ontology, task files), it has the concept of agent, with an instance of itself, the concept of skill with the corresponding instances (created by defskill or make-skill), the concept of goal with the corresponding instances (created by defgoal or make-goal). Those specific classes must be created before the skills and goals are defined, thus when executing the defagent macro or the make-agent function.

Thus, when a new agent is loaded in its own package we call the moss::%create-new-package-environment function to create an ontology stub. Assuming the agent is called ADDRESS, the function creates the following objects in the agent package:

- a proxy for the MOSS system: $E-SYS, system ID
- a local system name ADDRESS-MOSS-SYSTEM (entry point)
- an instance hosting the local environment: $E-SYS.1
- a local name for the instance: ADDRESS-MOSS
- a local class proxy for methods: $E-FN
- a local version of *none* and *any* classes and counters (for *none*)
- variables *moss-system* and *ontology* (synonym) with value $E-SYS.1

Now, in addition to that, we must create specific agent classes and instances. To do so, we call the omas::%create-moss-agent-classes agent method that adds:

- a class AGENT
- a class SKILL
- a class GOAL
- =summary methods for agent, skill and goal

Killing the Agent

When the agent is killed or crashes or commits suicide, nothing special happens. Next time around, all data structures are created anew when loading the agent.

10.3.4 Persistent Agents

A service agent with a persistent database is defined as:

```
(defagent :test :redefined t :persistency t)
```

This initializes the persistency slot of the agent structure to a non NIL value. While loading the rest of the agent files, the system checks if the database exists prior to loading the ontology. If the database does not exists the system issues a warning, creates it[4] and initializes the agent partition. It then reads the ontology files and uploads all object definitions into the database. The database stays open and the global entry

```
(OMAS-DATABASE *OMAS*)
```

is set to the pathname of the file OMAS-AGENTS.odb.

We have then two cases: (i) the agent is loaded for the first time and its partition in the database has not yet been created; and (ii) the agent was created, and is reloaded for some reason.

Determining whether the agent is loaded for the first time consists in checking: (i) if the OMAS-AGENTS.odb file exists and if the partition corresponding to the agent key exists. This is done by the moss::db-ready? function. If it exists, then we concluded that we are not loading the agent for the first time. Example:

```
(moss::db-ready? "OMAS-AGENTS" :ADDRESS *database-pathname*)
```

Loading an Agent for the first time

After creating the MOSS stub and agent classes like in the non persistent case, we must create the partition in the OMAS-AGENTS database, and save a minimal amount of data. Ontology and KB will be later saved after loading the corresponding files. When the OMAS-AGENT.odb file does not exit in the application folder, we create it, issuing a warning.

When saving the ontology, a special care must be taken concerning macros or service functions related to the ontology or the KB.

Loading a Recorded Persistent Agent

In that case, we simply bring the minimal number of objects into core from the database. When restarting an agent, e.g. after a crash, then we reload the agent[5], recreate all system objects, and then we must reconciliate such objects with the ontology concepts and KB objects on disk, regarding omas-agent and omas-skills classes and individuals, and entry-points.

Exiting the Agent Environment

If for some reason, we leave the agent environment (quit or crash), we must save some objects into the database.

10.3.5 Creating or Modifying Ontology or KB Objects

When creating or modifying objects of the ontology or the KB, the object base should be updated. The simplest way to do it is to keep a list of the objects that have been modified, and when the creation or modification is finished, to update the object base by overwriting existing objects. A specific slot

[4]Remember that it is a global database containing information for all the agents of an application executing in the same Lisp environment.

[5]Done by the load-agent function in the omas-W-init.lisp file.

of the agent structure indicates if a transaction in on-going or if each created or modified object must be saved without delay.

In practice, the list of modified objects may be stored in a particular property of the agent structure (to-save), and emptied when the saving is completed. To be safe, the objects to be modified should be saved first into a temporary flat file in case of crash during the operation (not done currently).

10.3.6 Saving Objects

An object is saved by sending it a =save message that does the adequate thing for each type of objects, namely:

- concepts are saved normally
- properties, entry points are saved normally, generic properties are checked for existence (this is probably not necessary)
- methods are saved together with corresponding functions
- nothing special for individuals, except that individual agents, skills or goals are not saved
- macros and functions, must be saved as text

10.3.7 Objects Not to Be Saved

Objects that we want not to save are marked with a (:no-save t) property on their p-list. For example, we do not want to save objects representing skills since they are recreated each time an agent is loaded[6].

10.3.8 First Time Saving

When saving for the first time, objects to be saved are obtained from the local *moss-system* object. All system objects are removed as well as system back pointers in entry points. Then, the whole lot is saved onto disk one object at a time.

10.3.9 Implementation for Next Versions

Next versions could be more sophisticated, e.g. working on text files, loading them and rebuilding them for concepts, and storing only the KB individuals.

10.4 Changes to the OMAS Code

10.4.1 Creating an Agent

defagent takes one more parameter: persistent

```
(defMacro defagent (name &key redefine hide learning package language context
                         version-graph master persistent)
  "creates an agent, puts it onto the *agents* list, name must be an unused symbol ~
      unless the agent is to be redefined with the option (:redefine t).Arguments:
   name: name given to the agent (e.g. MUL-3 or keyword :MUL-3)
   redefine: (key) if t allows the agent to be redefined
```

[6]One must be careful to cook up unique entry points for such objects. Otherwise the entry point is shared and saved, which will introduce an inconsistency in the database.

```
   hide: (key) if true, hide the agent from the user (defaul is nil)
   learning: (key) if t, allows the agent to remember results of tasks (default ~
      is nil)
   package (key): a keyword specifying the package for the agent name
      (if the agent name is a keyword).
   language (key): unused
   context (key): unused
   version-graph (key): unused
   master (key): applies to staff agents to specify the master PA
   persistent (key): specify that the agent has an object store for ontology and KB"
```

This has consequence onto the make-agent function:

```
(defun MAKE-AGENT (name &key redefine hide master learning (package *package*)
                          (language (or *language* :en))
                          (context (or moss::*context* 0))
                          (version-graph (or moss::*version-graph* '((0))))
                          persistency)
```

The agent structure must have one more slot: persistency. We add it to the self part of the structure.

10.4.2 Loading an Agent

To load an agent (AAA) we click on the LOAD button of the init window menu, which triggers the oi-load-on-click function that in turn calls the load-agent function[7]. The steps in the load-agent function are the following:

1. We load the agent file (e.g. :ADDRESS). At that time, the agent package is created and the make-agent function is called to create the agent structure. Here we have two cases: (i) the agent is created for the first time and the database does not exist; or (ii) the agent is being reloaded (e.g. after a crash or for a new session) and the database already exists. When the database does not exist, one must create it and create the agent MOSS environment. When the database exists, we must reinitialize the environment[8].

   When executing make-agent, the following actions are taken:

   - the agent name and key are created; the agent structure is created, initializing key, name, and persistency slots; the agent-being-loaded slot of the *omas* global parameters is initialized with a pointer to the agent structure;
   - the moss-make-agent function is called, which leads to the following steps:
     - an agent name (e.g. ADDRESS-AGENT) is created as well as the *moss-system* local variable i.e. in the agent package.
     - *if the agent is not persistent*, the %create-moss-classes-and-proxies is called. The following objects are created:
       (a) the MOSS-SYSTEM local class;
       (b) an instance of MOSS-SYSTEM;
       (c) the associated objects are recorded into the *moss-system* local object;
       (d) a counter for orphans, recorded into the *moss-system* local object;

[7]Defined in the W-init file).

[8]A possible approach would be to download the whole database at that stage, so that we have the ontology and KB in core, which simplifies the following processing. However, this approach has not been followed.

(e) a local class for methods linked to the moss::$FN class;

(f) a class for skills;

(g) a class for agents;

(h) summary methods for agents and skills;

(i) the agent object is created. The MOSS id of the agent is then inserted onto the p-list of the agent key (property :id) and returned.

– *if the agent is persistent and its database exists*, then and we open the database, load the agent MOSS object and the agent MOSS environment (*moss-system* defined in the agent package); initialize the base slot of the agent; let the base open; put the agent object id onto the p-list of the agent key; return the id. Because the set of skill may have been changed, we must re-create the skill concept; but we must not save the skill concept nor the skill instances, since they are reloaded each time the agent is re-loaded. The =summary method for the skills must not be saved. We can do that by putting the object ids onto the *objects-not-to-be-saved* list (see Section 10.4.2).

– *if the agent is persistent and the database does not exist,* we do like in the non persistent case.

When exiting the moss-make-agent function, the *moss-system* environment has been defined, and contains the minimal set of objects. Indeed, when loaded from disk the MOSS system entry-points have been stripped, since they are recreated each time OMAS is loaded.

- then, returning to the make-agent function, the agent structure is filled with the name of the ontology-package, language, context, version-graph, name of the agent is associated with the CLOS structure.
- Then the default skills are added, creating a new MOSS object for each skill, and linking it to the agent object.
- then the agent is inserted into the list of local agents, the main and scan processes are created, the agent window is refreshed and the make-agent function returns.

  At the end of the make-agent function, the agent CLOS structure has been initialized, the local *moss-system* contains either a minimal environment or the set of all local references including ontology and knowledge base (when loaded from the existing database), but still not the MOSS system entry-points.
- After executing the make-agent function, the rest of the agent file contains macros, functions, and user-defined skills that are added to the local *moss-system* environment.

After the agent file has been loaded, in case of a persistent agent being loaded for the first time, the database has not been created yet. This is done in the following steps of the load-agent function.

2. Once the agent file loaded, we must load the ontology file, unless we have a persistent agent with an existing database (in which case the database is opened and the base slot of the agent contains the handle to the database). When the database was already created, we need only add the MOSS system entry-points.

   In case of persistency[9] we face several cases:

   - If the database already exists, then open it and initialize the base slot of the agent structure by calling the load-initialize-environment function. The function does the following:

[9]To recognize if the agent is persistent or not, we use the agent-being-loaded slot of the *omas* global parameters that contains a pointer to the agent structure that has just been created. Thus, we can check the base slot of the agent structure.

    - loads and compiles variables;
    - loads and compiles macros;
    - loads and compiles methods;
    - loads and compiles service functions
    - merges the entry point list with MOSS entry points. Here the choice is to record all accessible entry points into the local *moss-system* environment.
  - if the agent is persistent but the database does not exist, we must create it, load the ontology file, and save the world[10].
    - we create the database with the load-agent-create-database function;
    - we load the ontology file with the moss::m-load function that allows forward references;
    - we save the world by using the moss =save-application-ojects-in-package method.
  - if the agent is not persistent, we simply load the ontology file.
  - finally, we merge the MOSS system entry points into the *moss-system* local environment using the load-agent-fuse-moss-entry-points function that adds the MOSS system entry points to the local environment entry points, updating the *moss-system* variable. It does not seem to matter that the resulting entry-point list in *moss-system* contains entries that are not accessible from the agent package.

3. Then for service agents we have no task nor dialog files.

One must realize that entry points contain references to all objects in the Lisp environment, regardless of their package.

Possible Problems

When loading an agent, we recreate the agent class, the agent instance, the model of skills and the instances of skills. However, in previous sessions the instance of the agent may have been modified, e.g. by adding more information concerning the agent acquired dynamically. On the other hand the newly create instance of agent is linked with the newly created skills. In addition the counter attached to the model of skills may not be right.

A possible way around this problem would be:

- to load the instance of agent, e.g. ADDRESS-AGENT in make-agent;
- not to save the skills nor the skill class and skill counter;
- to remove the skills from the instance of the agent when saving it.

This sounds a little complex. One could also:

- not save the skills nor the skill class and skill counter;
- use a profile object for storing all additional features of the agent;
- not save the agent, nor the moss-agent class.

An agent MOSS representation and associated skills should be recreated each time the agent is loaded. Thus, they should not be saved into the database...

Not saving objects into the database can be done by defining =save own-methods that do nothing. But we cannot save the =save method...

A better solution is to define a global *objects-not-to-be-saved* list and record the objects that we want to recreate each time (SKILL concept, SKILL counter, agent instance...).

[10]When saving the world at that stage, the *moss-system* local environment does not contain yet the system entry points.

Algorithm for saving ontology and KB

One first filters the application objects obtained from the local `*moss-system*` (by checking the package) and sends a =`save` message to all of them.

Algorithm for reconciliation

One gets all the entry-points from `moss::*moss-system*` and merge application information as read from the object base.

10.4.3 Creating or Modifying Ontology Concepts or Individuals

Creating or modifying ontology concepts can be done by MOSS functions or through the MOSS editor. We must tell MOSS to save the objects. When creating a class this could be done by adding a persistency option to the creation functions.

10.4.4 Accessing the Ontology and KB objects

When referring to the Ontology or KB objects (e.g. through get functions), they could be on disk. Thus, if an object key (ID) is unbound, one must try to load the corresponding object from disk before declaring an error. Such a check must be done in many functions and methods of the MOSS system.

10.5 Changes to the MOSS Code

10.5.1 Creating a MOSS object

A number of MOSS functions are creating new objects, namely:

- %make-concept
- %make-generic-tp
- %make-ep-from-mln
- %make-inverse-property
- %create-entry-point-method
- %make-prop-get-accessor
- %make-prop-set-accessor
- %%make-tp-from-ids
- %make-generic-sp
- %make-global-sp
- %%make-sp-from-ids
- %make-sp-raw
- %make-concept
- %make-virtual-concept
- %make-method

- %make-ep
- %build-method
- %make-ownmethod
- %make-universal
- %%make-instance-from-class-id
- %make-object
- %create-basic-new-object
- %create-basic-orphan
- =new methods (should not create objects directly, but use service functions)
- =new-version

We can do either one of two things;

- create a new object through a function that will take care of saving the object if needed
- add a function to save the newly created object.

The second option is easier to implement and requires only a key and object value to be saved on the edit list.

10.6 Loading a Persistent Agent

This section details the main function calls when a persistent agent is loaded.

10.6.1 First Time

The first time around the database (OMAS-AGENTS.odb) is not initialized. Either it does not exist, or the agent area does not exist. Remember the database is located in the application folder.

load-agent /cl-user

- get agent filename path
- set (agent-being-loaded *omas*)
- load agent file (e.g. AAA.lisp)
  - call make-agent /agent-package
  - create CLOS agent structure
  - call agent-set-moss-environment
    - set (persistency agent) to t
    - compute database pathname (OMAS-AGENTS.odb in application folder)
    - if database does not exist
      - create database if does not exist
      - record database pathname into (omas-database *omas*)

        · call moss::%create-new-package-environment, creating the MOSS stub in the agent package
        · call agent-create-moss-classes
          create OMAS-SKILL, OMAS-GOAL, OMAS-AGENT concepts
          mark concepts for not saving instances
          mark counters to be saved with value 1
          create instance of agent
          save MOSS instance id into (moss-ref agent)
    * if database exists
        · open database
        · if area does not exist
          create it
          record database pathname into (omas-database *omas*)
          call moss::%create-new-package-environment, creating the MOSS stub in the agent
          call agent-create-moss-classes (same as above)
          return
        · if area exists... (this does not happen the first time an agent is loaded)

- first time around: (persistency agent) is not :installed
- get agent-package from (ontology-package agent)
- compute ontology filename path
- load ontology using m-load, catching MOSS errors
- set package to agent package
- initialize moss::*database-pathname* from (omas-database *omas*)
- initialize moss::*application -package* from agent-package
- send a =save message to the *moss-system* local variable
- ... should insert a commit here
- reset (agent-being-loaded *omas*)

10.6.2 After the First Time

When the agent data have been saved into the database, loading the agent is done somewhat differently.

load-agent /cl-user

- get agent filename path
- set (agent-being-loaded *omas*)
- load agent file (e.g. AAA.lisp)
    - call make-agent /agent-package
    - create CLOS agent structure
    - call agent-set-moss-environment
        * set (persistency agent) to t

        - compute database pathname (OMAS-AGENTS.odb in application folder)
        - open database, area exists
            - reload MOSS environment, calling moss::db-init-app
            - recreate instance of agent being loaded: $E-OMAS-AGENT.1
            - record database pathname into (omas-database *omas*)
            - set (moss-ref agent) to $E-OMAS-AGENT.1
            - set (persistency agent) to :installed (to avoid loading ontology again)
            - return

- reset (agent-being-loaded *omas*)

Chapter 11

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. 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 12

OMAS vs JADE

Contents

This chapter describes contains a comparison between the JADE and the OMAS platform. It uses the JADE 3.1 platform as available in 2005 and should be updated with the new releases of the JASE platform.

12.1 Introduction

12.1.1 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 then comparing various features.

12.1.2 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 defined very easily. 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.

12.1.3 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, and once the goals are achieved the agent is terminated. Thus, an agent usually lives only until achieving some result.

The global approach in OMAS is different. Each agent is a separate entity. It is not an instance of a class. An agent, when created lives until it commits suicide, or somebody kills it. An agent has skills allowing it to perform services when asked. An agent may also have goals. However, when the goals are achieved, the agent does not terminate; it stays there, waiting for something else to do.

12.1.4 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 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 3 studies the question of message handling, i.e., how messages are selected and processed when received by an agent.
- Section 4 studies the problem of registering and discovering services.
- Section 5 studies the problem of content language.
- Section 6 studies the problem of goals and behaviours.
- Section 7 studies the Contract-Net protocol.
- Section 8 studies the problem of handling time.
- Section 9 studies the problem of concurrency
- Section 10 studies the problem of ontologies and how they are used in each platform.
- Section 11 compares the platform execution environment and the debugging tools.

12.2 Simple Approach to Problem 0

12.2.1 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 the initialization command line (or from the creation menu).

Each class is developed as a Java file and compiled with the JADE library. The resulting xxx.class files will be used by the JADE platform. Compilation produces a number of class files, one for each class and embedded classes.

OMAS

We create three separate agents, each with its own behavior. The resulting files are put into an application folder, called THOR-FAC where THOR 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 THOR-FAC folder contains a file names 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 cfsl for MCL (Macintosh) or fasl for ACL (Windows). Such folders are also contained in the THOR-FAC folder.

12.2.2 Programming the Multiply Agent

JADE

The Java code for the MULTIPLY agent class (named here MulAgentv0) is shown Figure 1. The class is an extension of the Agent class and contains several parts.

A setup function initializes the agent by adding a specific behaviour named MultiplyServer defined thereafter. A takeDown function is defined to be executed when the agent terminates.

The MultiplyServer inner class defines the MULTIPLY agent behaviour as cyclic. Whenever a message comes in, the agent reads it [myAgent.receive()] 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.

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;

import java.util.*;

public class MulAgentV0 extends Agent {
// Agent Multiply receives 2 numbers, multiplies them and returns the result

  // Put agent initializations here
  protected void setup() {
  // Add the behaviour serving queries from Fac agents
  addBehaviour(new MultiplyServer());
  }

  // Agent clean-up operations here
  protected void takeDown() {
  // Printout a dismissal message
  System.out.println("Mul1-agent"+getAID().getName()+" terminating.");
  }

/**
   Inner class MultiplyServer.
   This is the behaviour used by Mul agents to serve incoming requests
   Tales the incoming numbers and return an INFORM message with the answer.
 */
private class MultiplyServer extends CyclicBehaviour {
  public void action() {
  ACLMessage msg = myAgent.receive();
    if (msg != null) {
      // REQUEST Message received. Process it
      String argString = msg.getContent();
      ACLMessage reply = msg.createReply();

           int arg1,arg2;
           // recover args from string "arg1;arg2"
           arg1 = Integer.parseInt(argString.substring(0,argString.indexOf(";")));
           arg2 = Integer.parseInt(argString.substring(1 + argString.indexOf(";"),argString.le
           System.out.println("Mul-Agent; arg1: "+arg1+" ; arg2: "+arg2);

      reply.setPerformative(ACLMessage.INFORM);
              reply.setContent(Integer.toString(arg1 * arg2));
      myAgent.send(reply);
    }
         else {
              block();
         } // end message test
  }  // end action
}  // End of inner class MultiplyServer
```

OMAS

The OMAS Lisp code 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. This requires some obscure programming for compiling and loading properly (use of eval-when function).

The agent is defined by the omas::defagent macro. Its skills are defined in skill sections 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.

Auxiliary lines involving the defparameter form are not necessary. They are used by a manager agent to maintain this file automatically if needed.

```
;;;===============================================================================
;;;04/04/04
;;;                AGENT MUL-1
;;;
;;;===============================================================================

(eval-when (compile load eval)
  (unless (find-package :mul-1) (make-package :mul-1)))

;;; define name in the loading environment, so that it can be used without
;;; prefix, export it to the agent package

(in-package :mul-1)

(eval-when (compile load eval)
  (use-package :omas)
  (export '(mul-1))) ; reexport it for use in other packages

(omas::defagent mul-1)

;;; ============================== skill section ==============================

(defParameter *mul-1-multiply-functions*
  '((:functions static-multiply)))

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

(defun static-multiply (agent environment n1 n2)
  (declare (ignore environment))
  (sleep (1+ (random 2))) ; slow down execution so that we can see something
  (static-exit agent (* n1 n2))) ; return result to caller

:ENV

#+MCL (in-package "COMMON-LISP-USER")
#+ACL (in-package "COMMON-GRAPHICS-USER")

(eval-when (compile load eval)
  (use-package :MUL-1))

;;; ================================ environment ==============================

(defParameter *mul-1-environment*
  '((:skills :multiply)))
```

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.

  JADE agents are transient by default, unless a specific behaviour keeps them alive. OMAS agents are persistent 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.

- *Messages*

  The JADE agent explicitly retrieves the message [myAgent.receive()]. Retrieving messages in OMAS is done automatically.

  The JADE agent extraction process returns any message addressed to the agent[1], thus we assume that the messages are MULTIPLY requests and are well formatted. The OMAS system handles the selection process, calling the skill only when needed. 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()][2]. The OMAS agent gets the arguments of the message as the last arguments of the skill.

[1]We will see later that a message pattern can be used to retrieve only selected types of messages.
[2]We will see that an ontology can be used to transform the content into a Java object.

- *Processing the Message*

  The JADE agent executes the behaviour in the single agent thread. Thus, it is necessary for the agent to let other behaviours work by executing a [block()] instruction. However, the following incoming message will wake up all behaviours. The OMAS agent does not have this problem in the sense that the skill 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.

- *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 makes programming somewhat more obscure to people not familiar with packages.

12.2.3 Programming the Factorial Agent

The FACTORIAL agent is more complex than the MULTIPLY agent.

JADE

The Java code for the FAC agent is given Figure 3. It differs from the code for the MULTIPLY agent as follows.

We set up a list of known MULTIPLY agents in the mulAgent Table.

The setup function gets the FACTORIAL argument from the command line. We thus need to decode it, and if nothing is there, we quit.

The most important change is in the Behaviour. The function is called initially and then again, whenever the agent receives a message. The first time we call the function, we check if `nn` (the target factorial to compute) is less than or equal to 2. If so we are finished and quit. Otherwise we send a REQUEST message to one of the MULTIPLY agents picked randomly. We update `nn`. When the answer comes back, unless `nn` is 1, we send a REQUEST to multiply the returned result with `nn` to one of the MULTIPLY agents picked randomly. When we are finished, we print the result and set `step`=2, which terminates the FACTORIAL agent (through the `done()` function that returns true).

```
import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;

public class FacAgentV0 extends Agent {
/* Fac agent reads a number from the input and then tries to compute its factorial
     by subcontracting multiplies to the multiplying agents */

  // The list of known multiplying agents
  // This means that fac must be loaded after mul1 has been installed...
  private AID[] mulAgents = {new AID("mul0", AID.ISLOCALNAME),
                             new AID("mul1", AID.ISLOCALNAME)};
  int nn;

  // Put agent initializations here
  protected void setup() {
  // Printout a welcome message
  System.out.println("Hello! Fac-agent "+getAID().getName()+" is ready.");
  // Get the factorial number as a start-up argument
  Object[] args = getArguments();
  if (args != null) {
    nn = Integer.parseInt((String)args[0]);
    System.out.println("Fac; factorial to compute is: "+nn);

          // Perform the request
    this.addBehaviour(new RequestPerformer());
  }
  else {
    // Make the agent terminate
    System.out.println("No number was specified");
    doDelete();
  } // end test of the initial argument
}  // end setup

  // Put agent clean-up operations here
  protected void takeDown() {
  // Printout a dismissal message
  System.out.println("Fac-agent "+getAID().getName()+" terminating.");
  }

/**
   Inner class RequestPerformer.
   This is the behaviour used by Fac agents to compute factorials.
           When nn <= 2 return nn;
           Otherwise, send nn and nn-1 to be multiplied by the first MulAgent
           and set nn to nn-2.
           Wait for answer.
           When result comes back if nn <= 1, then pront the result,
   Otherwise, send result and nn to be multiplied, and set nn to nn-1.
 */

private class RequestPerformer extends Behaviour {
       private int result;  // The result returned by mulAgent
       private int step = 0;
       private int agentNb = 0;  // random index of agent to be called
       private String contentString;

  public void action() {
    switch (step) {
    case 0:
             // test the value of nn; when 2 or less, return
             if ( nn <= 2 ) {
                System.out.println("Result is: "+nn);
                // give up
                myAgent.doDelete();
              }
              else {
```

```
        // Ask first agent to multiply
        ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
                // Prepare message content
                contentString = new String(Integer.toString(nn--)+";"+Integer.toString(nn--));
                agentNb = (int) Math.floor(Math.random() + 0.5);
                System.out.println("Factorial; sending: "+contentString+" to mul"+agentNb);

                request.setContent(contentString);
                request.addReceiver(mulAgents[agentNb]);  // send to first agent of the list

        request.setConversationId("factorial");
        request.setReplyWith("request"+System.currentTimeMillis()); // Unique value
        myAgent.send(request);
        step = 1; // second time around we go to next block
              }
      break;

    case 1:
      // Receive result from multiply agent
      ACLMessage reply = myAgent.receive();
      if (reply != null) {
        // Reply received
                System.out.println("Factorial; received a result: "+reply.getContent());
                // check whether we are through
        if (nn <= 1) {
                   // We are through print fimal result and quit
                   System.out.println("Final result: "+reply.getContent());
                   myAgent.doDelete();
                   step = 2;
                }
                else {    // not finished yet
        ACLMessage newRequest = new ACLMessage(ACLMessage.REQUEST);

                contentString = new String(reply.getContent()+";"+Integer.toString(nn--));
                agentNb = (int) Math.floor(Math.random() + 0.5);
                System.out.println("Factorial; sending: "+contentString+" to mul"+agentNb);

        newRequest.setContent(contentString);
             newRequest.addReceiver(mulAgents[agentNb]);  // send to first agent of the list
        newRequest.setConversationId("factorial");
        newRequest.setReplyWith("request"+System.currentTimeMillis()); // Unique value
        myAgent.send(newRequest);
        }  // end if
              }  // message test
              else {
        block();
              } // end of message test
      break;

    } // end of switch
```

```
  } // end of action

  public boolean done() {
    return (step == 2);
  }

}  // End of inner class RequestPerformer
}
```

OMAS

The OMAS agent does the same thing as the JADE agent. The difference with the MULTIPLY skill is now that the :dumb-fac skill has two parts (a static part and a dynamic part. 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).

When the final result is obtained, the skill exits through the API dynamic-exit function, which returns the result to whoever asked for it.

```
;;;===============================================================================
;;;04/07/22
;;;               AGENT FAC
;;;
;;;===============================================================================

(eval-when (compile load eval)
  (unless (find-package :fac) (make-package :fac)))

;;; define name in the loading environment, so that it can be used without
;;; prefix, export it to the agent package

(in-package :fac)

(eval-when (compile load eval)
  (use-package :OMAS) ; use keyword to avoid creating OMAS symbol in package FAC
  (export '(fac))) ; reexport it for use in other packages

(omas::defagent FAC :redefine t)

;;; ================================ skill section ================================

(defParameter *fac-dumb-fac-functions*
  '((:functions static-dumb-fac
                dynamic-dumb-fac)))

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

(defun static-dumb-fac (agent environment 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
      (let (env)
        ;; 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)!
        (setq env (append (list :n (- nn 2)) environment))
        ;; record environment
        (update-environment agent env)
        ;; quit
        (static-exit agent :done))))

(defun dynamic-dumb-fac (agent environment 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 (cadr (member :n environment))))
    ;; 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 (nth (random 2) '(:MUL-1 :MUL-2))
                      :action :multiply
                      :args (list answer nn))
        ;; update environment
        (setf (cadr (member :n environment)) (1- nn))
        (update-environment agent environment)
        ;; then return an answer
        answer
        ))))

#+MCL (in-package "COMMON-LISP-USER")
#+ACL (in-package "COMMON-GRAPHICS-USER")
 (eval-when (compile load eval)
  (use-package :FAC))

;;; ================================= environment ==============================
(defParameter *fac-environment*
  '((:skills :dumb-fac)
    ))

:EOF
```

Comparing JADE and OMAS

The same remarks that were made for the MULTIPLY agent apply here. We can add the following.

- Initial Argument

  The number specifying the factorial to compute is given in the command line in the case of the JADE agent. For the Lisp agent this number will have to be provided by a message, which can also be done with JADE by using the Dummy Agent of the debugging interface.

- Agent Life Cycle

  Once it has computed the requested factorial, the JADE agent terminates. The Lisp agent remains alive and waits for another factorial to compute. Of course the same behaviour could be imposed to the JADE agent by setting up a cyclic behaviour, as was done for the MULTIPLY agents.

- Program values

  Keeping the value of nn between calls to behaviour or skill is done through the private space of the FACTORIAL agent for the JADE approach and could be declared as a global variable in the agent unique package for OMAS. However, for OMAS, it was thought that a specific area attached to a specific skill (task) and called the environment was more suitable for doing this. The environment is passed as an argument to the skill functions.

- Processes

  The JADE agent has a single thread of execution. 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.

12.2.4 Launching the Platform

First one must launch the corresponding platform before loading the agents.

JADE

For example on a Macintosh OSX one first specifies the path where the system can find the Jade library of classes, e.g. by setting a global variable:

```
set CLASSPATH = ':/Library/Home:/Library/Java/Home/lib:.:lib/jade.jar:lib/jade
Tools.jar:lib/Base64.jar:lib/iiop.jar'
```

Then one launches the GUI as follows:

```
java -cp $CLASSPATH jade.Boot {gui
```

Something like the following lines is printed on the terminal.

```
[1] 398
[Ordinateur-de-Jean-Paul-Barth-s:~/Applications/jade] barthes%     This is JADE 3.1 - 2003/12/
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.cselt.it/

IOR:000000000000001149444C3A464950412F4D54533A312E300000000000000001000000000000005400010100000
Agent container Main-Container@JADE-IMTP://Ordinateur-de-Jean-Paul-Barth-s.local. is ready.
```

... and the Jade GUI appears (Figure 12.1).

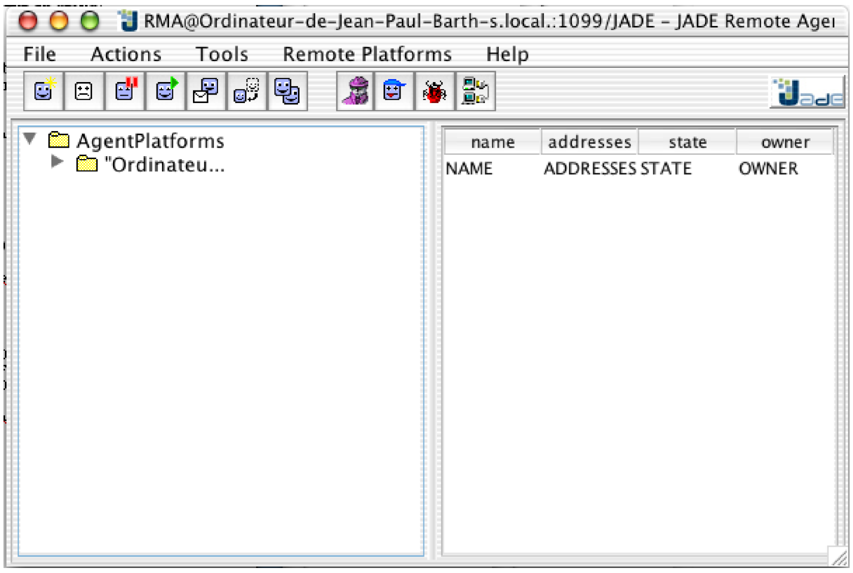
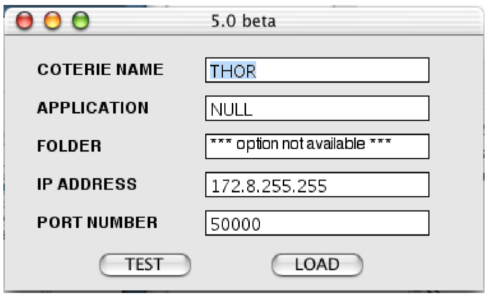


Figure 12.1: The JADE GUI

OMAS

The OMAS platform is launched by starting the Lisp environment and loading the load-omas-moss.lisp file for the Macintosh or 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. 12.2).



COTERIE NAME	THOR
APPLICATION	NULL
FOLDER	*** option not available ***
IP ADDRESS	172.8.255.255
PORT NUMBER	50000

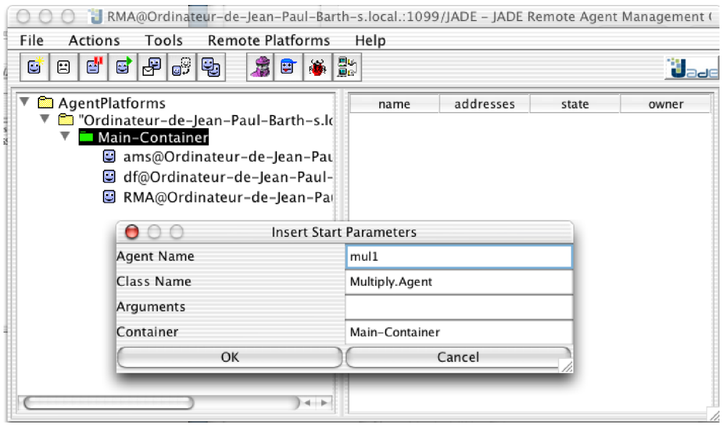
Figure 12.2: The OMAS Initial Window

IP address and port number are the local broadcast address and the port used by the platform.

12.2.5 Loading New Agents

JADE

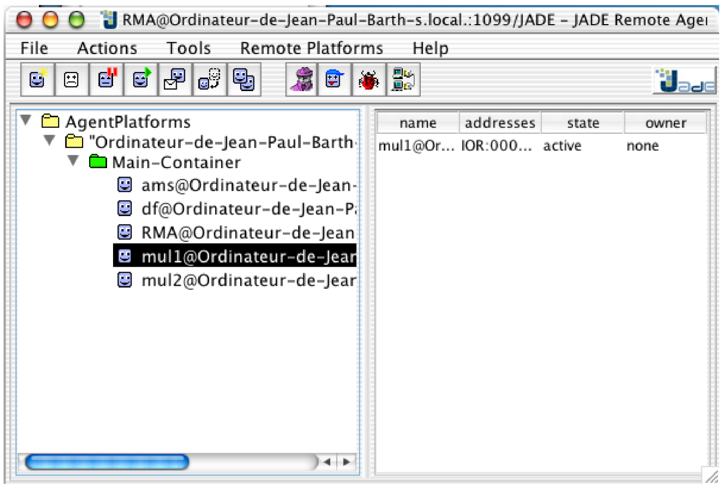
Jade agents are created one by one using the create agent button of the interface (Fig. 12.3). They can also be loaded using a batch file.


RMA@Ordinateur-de-Jean-Paul-Barth-s.local.:1099/JADE - JADE Remote Agent Management
File
Actions
Tools
Remote Platforms
Help
AgentPlatforms
"Ordinateur-de-Jean-Paul-Barth-s.lo
Main-Container
ams@Ordinateur-de-Jean-Pau
df@Ordinateur-de-Jean-Paul-
RMA@Ordinateur-de-Jean-Pa
name
addresses
state
owner
Insert Start Parameters
OK
Cancel

Agent Name	mul1
Class Name	Multiply.Agent
Arguments	
Container	Main-Container

Figure 12.3: Creating a new MULTIPLY agent named mul1

Fig. 12.4 shows the state of the container when two Multiply agents have been created.



name	addresses	state	owner
mul1@Or...	IOR:000...	active	none

Figure 12.4: Main container with two multiply agents

OMAS

In OMAS the 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. 12.5).

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.

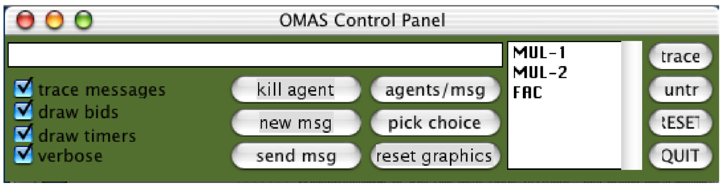


Figure 12.5: OMAS Control Panel for the FAC application

Another point is that if a Jade class file does not exist, an exception is raised and the program quits.

12.2.6 Executing Agents

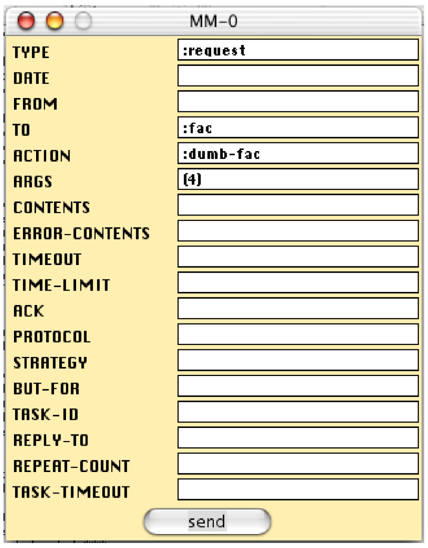
JADE

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, prints the result and quits.

OMAS

OMAS agents become active as soon as they are loaded. However, 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))
```


MM-0

TYPE	:request
DATE	
FROM	
TO	:fac
ACTION	:dumb-fac
ARGS	(4)
CONTENTS	
ERROR-CONTENTS	
TIMEOUT	
TIME-LIMIT	
ACK	
PROTOCOL	
STRATEGY	
BUT-FOR	
TASK-ID	
REPLY-TO	
REPEAT-COUNT	
TASK-TIMEOUT	

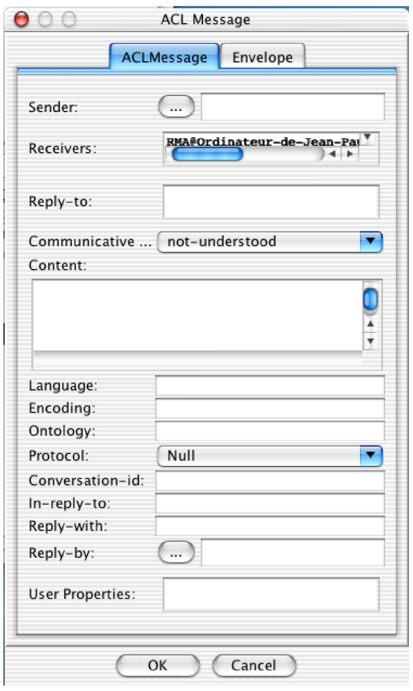
Figure 12.6: Message composition window

Once the message is sent and the factorial is computed, the Factorial agent stays active, waiting for another message or something else to do.

12.2.7 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. 12.7 to be compared with that of Fig. 12.6 for OMAS



Sender:
Receivers: RMA@Ordinateur-de-Jean-Pa
Reply-to:
Communicative ...: not-understood
Content:
Language:
Encoding:
Ontology:
Protocol: Null
Conversation-id:
In-reply-to:
Reply-with:
Reply-by:
User Properties:

Figure 12.7: 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. 12.8).

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.

12.3 Handling Messages

The next point to be studied is how the two platforms handle the messages, i.e. how an agent recovers and processes a message.

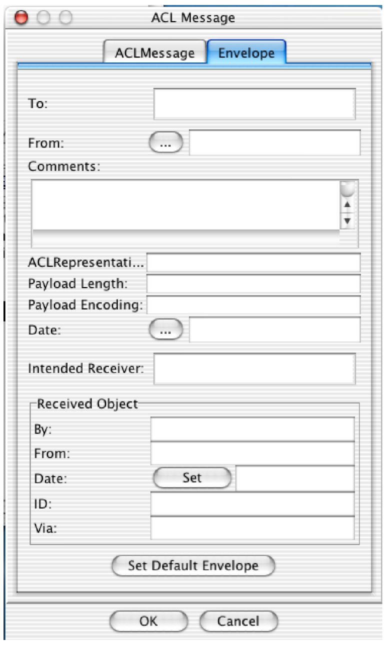


Figure 12.8: JADE message envelope specification window

12.3.1 Receiving Messages

JADE

Recovering messages for a JADE agent is accomplished very simply by invoking the myAgent.receive() function inside a behaviour.

One problem is that each JADE agent has a single thread of execution. Consequently, a behaviour executes until it relinquished control willingly. Thus, to avoid locks by polling constantly, JADE agents can use the block() function to block a given behavior and let others take control of the execution. All blocked behaviours go to sleep and will be awaken when a new message comes in. The order in which messages are processed must then be specified.

JADE Message Patterns In order for a specific behaviour to extract the proper message, the JADE programmer can use a system of patterns called *templates*.

```
private class MultiplyServer extends CyclicBehaviour {

  public void action() {

MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);

  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.

```
      ...

      // Prepare the template to get results
```

```
    mt = MessageTemplate.and(MessageTemplate.MatchConversationId("factorial"),

                                     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 not limited. 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.

12.3.2 Recovering the Message Content (simple case)

JADE

In the simplest case, a JADE agent can recover a message by using the getContent() function on the received message. It then has a simple string of data to be processed within the behaviour.

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.

12.4 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 a yellow page to find what agents are providing the service, which is the JADE approach, or sending a broadcast message, which is the OMAS approach.

12.4.1 Service Registration

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.

```
protected void setup() {

    // Register the multiplying service in the yellow pages

    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());
    ServiceDescription sd = new ServiceDescription();
    sd.setType("multiplying");
    sd.setName("JADE-multiplying");
    dfd.addServices(sd);

    try {
      DFService.register(this, dfd);
    }

    catch (FIPAException fe) {
      fe.printStackTrace();
    }
   ...
```

The type of the service is indicated by the "multiplying" string, the name of the service is "JADE-multiplying". 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.

```
          // Get the list of multiplying agents
          DFAgentDescription template = new DFAgentDescription();
          ServiceDescription sd = new ServiceDescription();
          sd.setType("multiplying");
          template.addServices(sd);

          try {

          //DFAgentDescription[] result = DFService.search(myAgent, template);
          // cannot use myAgent if not inside a behaviour ??

             DFAgentDescription[] result = DFService.search(this, template);
          System.out.println("Found the following multiplying agents:");
             mulAgents = new AID[result.length];
             for (int i = 0; i < result.length; ++i) {
                 mulAgents[i] = result[i].getName();
           System.out.println("   "+mulAgents[i].getName());
             }
          }
```

```
catch (FIPAException fe) {
  fe.printStackTrace();
}
...
```

12.4.2 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 answer.

JADE

JADE has no particular mechanism for sending a broadcast, other than sending the message to all agents after obtaining the names of the agents from the DF service. However, this is not 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 their answer.

OMAS

In the case of OMAS, it is very simple to send a broadcast 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 for a given time for possible answers and process them using two 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 case answers are collected until a timeout occurs or until we have the number of specified answers, in the last case answers are collected until a given time elapses, and 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 but 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 be able to talk to distant agent OMAS uses a special transfer agent that may be viewed as a gateway.

12.5 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.

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[3] :

```
(Person :name Giovanni :age 33)
```

JADE needs to transform the input string into an instance of a person class:

```
class Person {
   String name;
   int age;

   public String getName() {return name;}
   public void setName(String n) {name = n;}
   public int getAge() {return age;}
   public void setAge(int a) {age = a;}
   ...
}
```

The instance should be initialized with:

```
name = \Giovanni";
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 either as a string or as a list. 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 used as such, regardless of the existence of a class person.

[3]In fact this is not true and JADE agents could use introspection to recover the names of the existing classes. This does not seem to be used by the JADE developers.

12.6 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 concurrently. 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() {
  // Printout a welcome message
  System.out.println("Hello! Fac-agent "+getAID().getName()+" is ready.");

  // Get the factorial number as a start-up argument
  Object[] args = getArguments();
  if (args != null) {
    nn = Integer.parseInt((String)args[0]);
    System.out.println("Fac; factorial to compute is: "+nn);

          // Perform the request
    this.addBehaviour(new RequestPerformer());
  }
  else {
    // Make the agent terminate
    System.out.println("No number was specified");
    doDelete();
  } // end test of the initial argument
}  // end setup
```

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. Note that for skills requiring to subcontract part of the task, that 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.

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)
```

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)))
                      :task-id :BUY-BOOK
                      )))
```

Thegoal-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
```

12.6.1 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, since JADE agents are created and live only for the duration of a goal, which is not the case for OMAS agents.

Triggering conditions (activation on a specific date or waiting delays) seem to be easier to specify in the OMAS syntax.

12.7 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.

12.7.1 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) solicits 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. Ir 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.

12.7.2 OMAS Contract-Net

The Contract-Net protocol is implemented by OMAS by specifying the protocol parameter in the send-subtask API function. E.g.

```
;; if a multiply agent is timed out, then reallocate to another one
;; by simply doing another broadcast
(send-subtask agent :to :ALL :action :multiply
              :args (omas::args message)
              :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.

```
;; if a multiply agent is timed out, then reallocate to another one
;; by simply doing another broadcast
(send-subtask agent :to :ALL :action :multiply
              :args (omas::args message)
              :protocol :contract-net
              :strategy :collect-answers)
```

For example the BOOK example application of the JADE manual could use the following code:

```
(defskill :REQUEST-PERFORMER BOOK-BUYER
  :static-fcn static-REQUEST-PERFORMER
  :dynamic-fcn dynamic-REQUEST-PERFORMER
  :select-best-answer-fcn select-best-answer-REQUEST-PERFORMER
  )

(defun select-best-answer-request-performer
       (agent environment answer-message-list)
  "keep the cheapest offer"
  (declare (ignore agent environment))
  ;; order the collected answer messages by increasing prices
  ;; we do that only when there is more tan one answer
  (when (cdr answer-message-list)
    (setq answer-message-list
         (sort answer-message-list #'<= :key #'omas::content)))
  ;; select the message corresponding to the lowest price
  (omas::contents (car answer-message-list))
```

12.7.3 Comparison JADE/OMAS

Both platforms implement the Contract-Net protocol with the same possibilities. The OMAS approach seems to be much simpler to implement, the system performing all the bookkeeping automatically.

Another difference is that the OMAS manager does not have to know what are the other agents on the platform.

12.8 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.

12.8.1 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 has to program a 10 second delay after the agent has been created:

```
Public class MyAgent extends Agent {
  protected void setup() {
    addBehaviour(new.WakerBehaviour(this, 10000);
      protected void handleElapsedTimeout() {
        // perform operation, e.g. go buy a book
   myAgent.addBehavior(new RequestPerformer());
      }
    } );
  }
  // make the agent terminate
  doDelete
```

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.

12.8.2 OMAS Delays

OMAS delays are specified at the goal level 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.

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).

12.8.3 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."

12.8.4 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.,

```
  ;; ask about the prices of the book
  (send-subtask  agent :to :all
                 :action :offer-request
                 :args (list title)
                 :timeout 10   ; allow 10 seconds to answer
                 )
```

In practice any message can take a timeout parameter. The default behavior is to send the subtask two more times and quit, aborting the main task if nothing return 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.

```
(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)
```

Of course one needs to write the corresponding function:

```
(defun timeout-fast-fac-with-timeout (agent message)
  "function for handling timeout errors, rescheduling multiply subtask as a ~
subtask using a contract-net protocol."
  (case (omas::action message)
    (:multiply
     ;; if a multiply agent is timed out, then reallocate to another one
     ;; by simply doing another broadcast
     (send-subtask agent :to :ALL :action :multiply
                   :args (omas::args message)
```

```
                    :protocol :contract-net
                    :timeout *multiply-timeout*)
     ;; exit
     :done)
   ;; for any other subtask type we let the system process the timeout condition
    (otherwise
     :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.

12.8.5 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. In practice it would not be very useful for most agents, since they die after having executed their behaviours.

12.8.6 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.

12.8.7 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 are transient and live only to accomplish a specific goal.

12.9 Executing Several Tasks Concurrently

The execution mechanism is quite different for JADE and for OMAS.

12.9.1 JADE Concurrency

For different reasons including easy migration to another machine, JADE agents execute 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. 12.9.

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, blocking it until a new message arrives.this is done as follows:

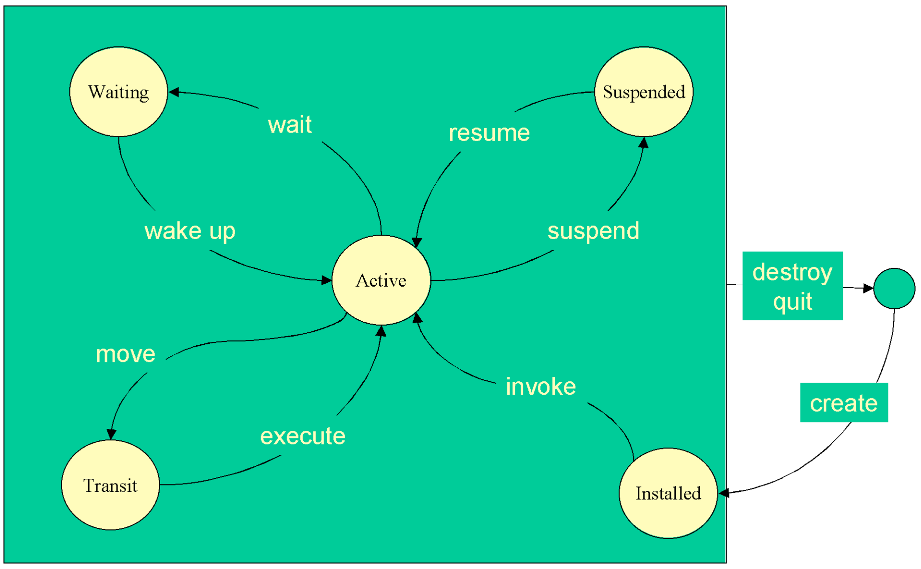


Figure 12.9: Different states of a JADE agent

```
public void action() {
  ACLMessage msg = myAgent.receive();
  if (msg != null) {
    // Message received. Process it.
    ...
  }
  else {
    block();
  }
}
```

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.

12.9.2 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 bides), then it is executed immediately by the agent scan process. 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 task it can execute in parallel.

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.

12.10 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.

12.10.1 JADE Ontologies

A JADE ontology must contain a definition of all the objects and properties that can be handled by the agents. However, since the 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.

12.10.2 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.