CONTENTS

1 Introduction ................................................................. 5
  1.1 OMASWA ................................................................. 5
  1.2 Common Lisp ............................................................ 5
  1.3 Related documents ..................................................... 5

2 OMAS Static Structures ............................................. 6
  2.1 Agent Structure ......................................................... 6
  2.2 Message Structure ..................................................... 6

3 OMAS-WA Processes ............................................... 8
  3.1 Agent Processes ........................................................ 8
  3.2 Agent States and Transitions ....................................... 9
  3.3 Personal Assistant Agents ......................................... 9

4 OMAS Execution Environment ................................ 11
  4.1 Control Panel .......................................................... 11
  4.2 Agent Windows ........................................................ 12
  4.3 Graphics Trace .......................................................... 12
  4.4 Textual trace .............................................................. 12
  4.5 Master’s Interface ...................................................... 13

5 Defining Agents ....................................................... 14

6 Defining Skills ......................................................... 15

7 The Agent Language ............................................... 16

8 Global Variables ...................................................... 20

9 Basic OMAS Mechanisms .......................................... 21
  9.1 Processing a :request Message .................................... 21
  9.2 Processing an :answer Message .................................... 22
  9.3 Processing an :abort Message ....................................... 23
  9.4 Processing a :cancel Message ......................................... 23
  9.5 Processing an :inform Message ....................................... 24
  9.6 The Time Limit Mechanism .......................................... 24
  9.7 The Timeout Mechanism .............................................. 24
  9.8 Contract-net ............................................................. 25
  9.9 Processing a :call-for-bids Message ............................. 26
  9.10 Processing a :bid Message ............................................ 26
  9.11 Processing a :grant or :cancel-grant Message .................. 27
  9.12 Processing a :soft-request Message .............................. 28

10 Global Variables (file: globals) ................................. 30
  10.1 Various Windows .................................................. 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.3</td>
<td>Service Functions</td>
<td>60</td>
</tr>
<tr>
<td>20</td>
<td>Processes (file: processes)</td>
<td>64</td>
</tr>
<tr>
<td>21</td>
<td>A.P.I. (file: API)</td>
<td>66</td>
</tr>
<tr>
<td>21.1</td>
<td>Macros</td>
<td>66</td>
</tr>
<tr>
<td>21.2</td>
<td>User and Scheduler Functions</td>
<td>66</td>
</tr>
<tr>
<td>22</td>
<td>Control Panel (file: control-panel)</td>
<td>71</td>
</tr>
<tr>
<td>22.1</td>
<td>Classes and Methods</td>
<td>72</td>
</tr>
<tr>
<td>22.2</td>
<td>Various Functions</td>
<td>72</td>
</tr>
<tr>
<td>23</td>
<td>Agent Windows (file: agentwin)</td>
<td>74</td>
</tr>
<tr>
<td>23.1</td>
<td>Global Variables</td>
<td>74</td>
</tr>
<tr>
<td>23.2</td>
<td>Classes and Methods</td>
<td>74</td>
</tr>
<tr>
<td>23.3</td>
<td>Functions</td>
<td>75</td>
</tr>
<tr>
<td>24</td>
<td>Graphics Trace (file: graphics)</td>
<td>76</td>
</tr>
<tr>
<td>24.1</td>
<td>Classes and Methods</td>
<td>76</td>
</tr>
<tr>
<td>25</td>
<td>Proxy Windows (file: proxywin)</td>
<td>80</td>
</tr>
<tr>
<td>25.1</td>
<td>Structure and Methods</td>
<td>80</td>
</tr>
<tr>
<td>25.2</td>
<td>Functions</td>
<td>80</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 OMASWA

OMAS-WA is derived from previous multi-agent environments, in particular from OMAS, OSACA, and from SMAS.

OSACA (Scalabrini) is a complex distributed multi-agent platform developed in a UNIX environment. As such it is powerful, but too bulky and too slow. SMAS (Barthès) is a simulation environment developed to design and test complex agent implementations. It runs in a CommonLisp Macintosh environment. OMAS is the counterpart of SMAS in the sense that SMAS agents developed and tested in the SMAS environment can be run in the OMAS environment. OMAS is implemented using the multiprocessing facilities of the Macintosh CommonLisp (MCL from Digitools) and runs in real time. It is not yet distributed over a network, but this should be implemented soon. OMAS-WA is a new version of OMAS implementing personal assistants (OMAS-WA = OMAS With Assistants). OMAS-WA v2 also allow distributed execution on several machines using a UDP transport protocol.

In all the mentioned systems (OSACA, SMAS, OMAS), agents are totally independent but belong to a cluster called a “coterie.” Communications occur thanks to several protocols (basic and a modified version of Contract-Net). The protocol is specified at the message level and protocols can be mixed. Messages are sent in three modes: point-to-point, multicast, or broadcast. Agents are cloned from a generic agent and given skills. Agents can dynamically enter or leave the group. In the simulation system, SMAS, scenarios are used to specify the simulation sessions.

The current document describes the agent structure, how an agent is implemented, and the execution environment. The main function in the various modules are commented. As such, it replaces the OMAS v0 Reference Manual.

1.2 Common Lisp

Please refer to


1.3 Related documents

Other related useful documents are:

- N130-SMAS 2.0 Primer
- N132-SMAS 2.0 Ref
- N138-SMAS-AUCTION
OMAS Static Structures

OMAS static structures comprise agents and messages.

2.1 Agent Structure

An OMAS agent has a rather complex structure, implemented as sub-objects:

- **appearance**, controlling the graphics window and the master’s window
- **comm**, containing information about the communication buffers (input-messages buffer, and input- and output-log)
- **control**, containing information about the work being done (agenda, status, task-in-progress, pending-bids, saved-answers, saved-bids, last subtask result)
- **self**, contains internal information about the agent (features indicating for example if the agent is a learning agent, memory, user’s structured data, goals, intentions)
- **skills**, contains the list of skills to be applied
- **tasks**, contains a model of tasks or projects that are currently executed
- **world**, contains information about the state of the external world (acquaintances, environment, external services, i.e., services provided by other agents.
- **ontology**, containing the local ontology.

2.2 Message Structure

Static Structure

A message has the following structure:

- **action**, referring to the skill that is requested
- **args**, containing arguments necessary to execute the skill
- **but-for**, only in cancel-grant messages
- **contents**, contains the result of a task
- **date**, date at which the message was issued (universal time)
- **error-contents**, reason for an error
- **from**, agent that sent the message
- **protocol**, protocol being used in this message
- **repeat-count**, version of the message
- **reply-to**, agent(s) to which the answer must be sent
- **task-id**, identifier of the task
- **task-timeout**, timeout for the task (at the side of the sender)
- **time-limit**, limit set to execute the task
- **to**, agent to which the message is sent
- **type**, type of message (see hereafter)

Each message does not contain information for all the properties. It actually depends on the type of message being sent.

**Message Types**

Messages can be of the following type:
- **abort**, for aborting a task and sending an error message to the agent that requested the task
- **answer**, for sending an answer (can be an error)
- **bid**, for bidding as an answer to a call-for-bids
- **bid-with-answer**, bidding but sending the answer at the same time (e.g., in case it was found in the memory)
- **call-for-bids**, request for bids on a particular task
- **cancel**, abort a task but do not send an error message to the agent that has granted the task
- **cancel-grant**, grant a task to some agents and cancel it for others
- **grant**, grant a task to the agent receiving the message
- **request**, ask for a task to be executed
- **soft-request**, ask for a task to be executed if there is nothing better to do.

Colors are assigned to messages so that it is clearer to see what happens on the graphics trace.
3 OMAS-WA Processes

Processes use static structures to organize agent interaction. OMAS-WA processes are basically the same as OMAS processes except for processes dealing with a user interface in assistant agents. We’ll distinguish between service agents and Assistants that are service agents augmented with a user’s interface.

3.1 Agent Processes

Each OMAS agent, whether Service or Assistant, has a number of attached processes. The number changes during the life of the agent.

In particular an agent has two basic persistent processes:

− the scan process which continuously watches the input-messages box
− the mbox process which processes messages sent be the scan process that are relevant to the particular agent.

During the agent life other transient processes are created as needed. Such processes are timers (e.g., timeout for call for bids, timeout on a subcontracted task, time-limit on an executing task).

Input-messages arrive into the input-messages box. The scan process filters them out. It answers some of them and passes others to the agenda in order for the mbox process to execute them. Both processes are asynchronous and operate continuously.

The type of input messages that can arrive in the input-messages box can be:

− abort, that causes the agent to abort a task sending an error message to the agent that sent it
− answer, an answer to a subtask
− bid, an answer to a call for bids

Figure 3.1 - Processing messages
− *bid-with-answer*, an answer to a call for bids that contains the answer to the requested task
− *call-for-bids*, a call for doing some task
− *cancel*, that causes a task to be aborted, but no message is sent back to the requesting agent
− *cancel-grant*, a cancel message containing exceptions. A cancel-grant is an answer to a bid canceling granting the task to agents that won the bid and canceling it for other ones
− *grant*, a message granting a task after a bid
− *inform*, a message containing some information
− *request*, a request to do a particular task.

### 3.2 Agent States and Transitions

At any given time the agent can be in one of four different states, depending on what it is doing:

− *:idle*, meaning that the agenda is empty and the mbox process waits for some message to process
− *:busy* meaning that the agent is executing a local task
− *:waiting-for-answer*, meaning that the agent has subcontracted some tasks and is waiting for answers to such tasks
− *:busy-in-case*, meaning that the agent had nothing to do and is executing some task in case it could help somebody.

Note that an agent state does not depend on what messages are actually received but depend on what it is doing (or not doing).

Transitions occur when an agent has completed a task or when it starts a new one, or when some timer process kills the currently executed task.

### 3.3 Personal Assistant Agents

When the agent is an assistant a third persistent process is attached to the interfacing window with the master. This “master process” is in charge of the dialog with the user and also of transferring the user’s requests to the assistant agent (Figure 3.2).
Figure 3.2 - Main processes for a Personal Assistant
4 OMAS Execution Environment

The current execution environment contains a number of features, some of them inherited directly from the SMAS simulation environment. The environment is shown on the next figure.

4.1 Control Panel

A control panel can be used to specify some options, like showing a graphics trace involving some or all agents, displaying the messages, or resetting the system in case of error.
4.2 Agent Windows

Agent windows can be displayed to trace what messages are exchanged or processed by each agent.

![Agent windows for VENDOR and BUYER-1]

Figure 4.3 - Agent windows for VENDOR and BUYER-1

The left part of the agent window shows messages as they enter the input-messages box, and the right-hand side displays the content of the agenda. Most of the time however, messages are transient and it is quite difficult to track them down since they do not stay very long. The color of the left-hand side indicates the state of the agent.

4.3 Graphics Trace

The graphics trace is used to track exchanged message. The content of each message can be printed as shown on Figure 4.4.

![Graphics trace]

Figure 4.4 - Graphics trace

4.4 Textual Trace

A textual trace can be obtained by ticking the “verbose” square of the control panel, which prints a trace of various operations done on all agents. Alternatively only certain agents can be traced.
4.5 Master’s Interface

Still experimental. Not described here.
5 Defining Agents

A regular agent is defined using the defagent macro.

```
defagent name {options}
```

allows to define agents. The name argument must be unique, and the options can be used to allow the user to redefine agents (or to prevent that), or to hide an agent (presumably doing some system service).

```
?(defagent AGENT-1)
AGENT-1
```

After defagent is called, an agent is produced and starts living (i.e., its associated processes have been created and run).\(^1\)

Redefining agents

If the user wants an agent to be redefinable, then the option “:redefined t” must be added to the macro call, e.g.:

```
?(defagent AGENT-2 :redefine t)
AGENT-2
```

AGENT-2 can be redefined any number of times.

The global agent list

When an agent is defined, its name is appended to the list of already defined agents. The list is kept as the value of a global **agents** variable.

```
? *agents*
(AGENT-1 AGENT-2)
```

---

\(^1\) However, the agent is not automatically displayed in the graphics trace.
6 Defining Skills

Skills implement the expertise of an agent. Currently, this is done by defining a set of Lisp functions referenced in the \texttt{defskill} macro.

\begin{verbatim}
defskill skill-name agent-name &key static-fcn dynamic-fcn how-long-fcn how-long-left-fcn
\end{verbatim}

allows to define a new skill.

Simple Minimal Skill

The minimum for a skill is to have a static function. If the skill does not need help from other agents, then it is not necessary to set up a dynamic function. AGENT-1 knows how to multiply 2 numbers.

? (defskill MULTIPLY AGENT-1
    :static-fcn AGENT-1-multiply-static)

More Complex Skills

When an agent needs help from other agents to execute a task, then the skill is more difficult to design. Assume that AGENT-2 is an auctioneer selling goods, then its AUCTION skill will have an initial part when it announces a particular item to sell, and a dynamic part that will be called whenever some buyer agent will submit an offer.

? (defskill AUCTION AGENT-2
    :static-fcn static-auction
    :dynamic-fcn dynamic-auction)

The other options how-long and how-long-left functions are used to give an estimate of the time it will take for an operation to complete respectively before it is started and while it is in progress. Such function may be provided by the user, and if provided are used to compute offer delays when using the contract-net protocol.

All skills must be written using the Lisp language or interfaced using the foreign function mechanism if they are written in some other language. When using the Lisp language, one can use service functions that serve as an API. He functions are described in the following section.
7 The Agent Language

The agent language is a set of functions to be used as an API to the OMAS platform. They provide various services that are useful when writing functions implementing skills.

**abandon-current-task** agent &optional message function

When in the process of executing a particular task, the agent might decide to abandon the task, killing all subtasks and returning to the :idle state. This is useful when an agent has a particular skill but does not want to answer the request. The function kills and recreates the main process to keep a clean environment.

**answering-agent** agent function

Provides the name of the agent that sent an answer. Useful to know what agent, e.g., in the dynamic part of a skill.

**cancel-all-subtasks** agent &optional current-task function

Cancel all subtasks launched by a particular agent. Get them from the list of pending tasks contained in the task-frame of task-in-progress. Does not cancel the main task in progress.

**cancel-current-task** agent &key message no-subtasks function

Same function as abandon-current-task, but does not send error message to the agent that required the task.

If no task is being currently executed, then does nothing. Otherwise the calling function should make sure that task-in-progress is the same the one that was executing when the function was called; i.e., should call without-interrupts.

We do the following:
- cancel subtasks unless no-subtasks flag is set
- remove the pending-bids entries created by the task
- reset the main process if we are not in it
- remove possible repeated requests from the agenda
- remove the waiting-task description entry if the agent is learning
- reset the task-in-progress entry (remove the message)
- reset the status to :idle (displaying the result)

**Arguments**

agent: agent
message: (key) currently ignored
no-subtasks: (key) if t, indicate that no subtasks have been spawned and that if is not necessary to send cancel messages.

**cancel-answering-subtask** agent function
Cancel subtask corresponding to the received answer message being processed.

**cancel-subtask agent answer**

Cancel the subtask corresponding to subtask-id. Any info put into the user-managed environment area should be cleaned by the user. We clean input-messages, agenda, 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.

**Arguments**

agent: \( \text{agent} \)

subtask-id: \( \text{id of the task to be cancelled} \)

**create-subtask-id agent**

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.

**create-task-id agent**

Simply provides a task id. This version uses a global counter rather than gentemp so that we can reset task ids when debugging.

**dynamic-exit agent answer**

must terminate the dynamic part of a skill. Allows the system to do some cleanup.

**forget agent**

Used to wipe out agent’s memory.

**get-internal-time-string**

Produces a string with hr:mn:sec from internal time

**get-time-limit agent**

obtain the time-limit associated with executing task. If none, then returns positive integer in the system.

**pending-subtasks? agent**

Return the list of pending tasks. If processing an answer message, then list contains at least 1, the one corresponding to the answer being processed.

**purely-local-skill? Agent skill**

Checks if a skill operates locally, i.e., not spawning any subtask. This is verified when there is no dynamic part to the skills. Returns nil if agent does not have the skill.

**reset-all-agents**


Clear all agents, resetting input and output trays, and emptying tasks; resetting number of tasks to 0; and clock to 0.

**select-bet-bids agent, bid-list**

Select best bids in a list of bids according to job-parameters found in :contents. For now best bid is the earlier job.

**send-message message**

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
- If the target is ALL then we send the message to all agents known by the system
- If the target is :all-and-me we do the same as for :all including oneself
- If the target is a list of agents, then we send to each agent of the list
- If the target is a single agent, then we sent the message to the target.

*Arguments:*
- **agent:** agent
- **message:** message to send.

**send-subtask agent**

&key to action args (delay 0) (repeat-count 0) task-id timeout protocol

Prepares a message containing the parameters for issuing a subtask to another agent. Build a subtask-frame and add it to the subtask-list slot. Checks the :protocol variable and determines the output message accordingly.

*Arguments:
- **to:** to what agent(s) the task will be sent (aan associated value of ALL means broadcast)
- **action:** what skill is required
- **args:** arguments for the skill
- **delay:** additional delay to add up to the *subtask-creation-delay*
- **repeat-count:** by default 0 (normally not a user feature)
- **task-id:** specified if the user wants to give a name to the subtask. Otherwise the system makes up a name.
- **timeout:** specific timeout can be set
- **protocol:** allows to specify :contract-net as a possible protocol.

**static-exit agent answer**

Must be called when exiting from the static part of any given skill.

**update-environment agent environment**
Asks the system to update the value of the user-reserved environment area.
8  Global Variables

A number of global variables are used in the system. Among them:

*OMAS-window* nil "simulator control panel"

*graphics-window* nil "graphics trace window"

*agents* () "list of all active agents"

*agent-list* nil "list of agents to be displayed in the time graph"

*broadcast-strategy* :take-first-answer "specify broadcast /multicast strategy. Can be :take-first-answer or :collect-answers"

*default-call-for-bids-repeat-count* 3 "number of times we renew our wait when no bids present."

*contract-net-strategy* :take-first-answer "specify contract-net strategy. Can be :take-first-answer or :collect-answers"

*max-repeat-count* 2 "number of times a message will be repeated after a timeout"

*ontology* () "ontology for normalizing concepts: entries are a concept associated with generic concepts. Useful for normalizing skills"

*default-call-for-bids-timeout-delay* 30 “time to wait for an answer to a call-for-bids: 0.5 s"

*highest-time-limit* 60000000 "1000000 seconds a little less than 2 years"
9 Basic OMAS Mechanisms

The functioning of the OMAS platform follows a set of basic mechanisms that are reviewed in this Section.

A first mechanism and set of messages deals with regular requests and answers. A second mechanism deals with timeouts and time limits. A third mechanism and set of messages deals with the contract-net protocol.

9.1 Processing a :request Message

A request message is one of the most straightforward message that an agent can receive.

A simple request message is sent to a contractor agent. It arrives into the input-messages mbox of the agent. The agent has a special process called \textit{<agent-name>-scanner} that detects the incoming message.

\begin{itemize}
  \item since it is a :request the scanner transfers it to the agenda of the agent unless the result can be read from memory, in which case the answer is sent back to the caller readily.
  \item the agent has a main process that wakes up if anything is written to the agenda.
\end{itemize}

The main process thus calls the \textit{agent-process-agenda-task} function that is used to sort out the various types of messages found in the agenda. Since it is a request message, the \textit{agent-idle-process-request} function is called.

\begin{itemize}
  \item the function does the following
    \begin{itemize}
      \item sets the agent status to :busy
      \item checks whether the agent has the required skill. If not the message is ignored, the status reset to :idle and the process terminated.
    \end{itemize}
  \item the static skill is applied
    \begin{itemize}
      \item if the result is :abort, then the agent is reset and the task is aborted, the agent's state is reset to :idle
      \item otherwise the static skill was executed and we have two cases
        \begin{itemize}
          \item if the task is atomic, then we simply send back the answer, resetting the agent.
          \item if the task has spawned subtasks, then we check if the agent has a dynamic-skill function. If not, we declare an error. When we have a dynamic-skill function then we call the function \textit{agent-waiting-for-answer-process-answers}
        \end{itemize}
    \end{itemize}
  \item The function is a loop with a waiting condition for answers to the subtasks. While waiting, the agent state is set to :waiting-for-answer. Every time an answer comes back, we set the agent state to :busy, use the dynamic part of the skills to
\end{itemize}
process the answer, remove the subtask-frame from the
subtask-list of the task-frame of the task-in-progress. We then
return to waiting setting the agent state to :waiting-for-answer.

− When all subtasks have received an answer, then the dynamic
part of the skill should detect it and call the dynamic-exit function
which sends the result back to the calling agent or forward it to a
different destination agent (specified in the message), does some
clean up and exits. If the dynamic-exit task fails to detect the end
of the task, then the function will exit after all subtasks have
been executed and removed from the subtask-list of the task-
frame of the task-in-progress. However, in that case, no clean
up is done.

− The main process then returns to wait on the agenda box.

9.2 Processing an :answer Message

When an answer message arrives in the input-messages box, it is
immediately transferred into the saved-answers slot. Normally the agent is
in the waiting-for-answer state and is waken up by the arrival of the
answer. Waiting is performed by the function agent-waiting-for-answer-
process-answers

We do the following:

− set the status to :busy

− select a message to process by calling the function agent-select-answer-
message

− record the answer message being processed in the answer-being-
processed slot

− find the task-frame of the answer being processed in the processed-
subtask variable.

. if we do not find the corresponding task-frame, then since our agent
deals with one task at a time and the answer cannot be one to a yet
unprocessed task in the agenda, it can only mean that it is an answer to
an old task that was already processed. Thus, we can safely ignore it.

. otherwise,

− if we have a timeout on this subtask, then kill the timer and clean
up tables and, if the subtask was issued several times, then send a
cancel to the processed subtask.

Now depending on the subtask several cases can occur:

− the subtask was a simple request, in which case we call the
dynamic part of the skill to process it, using the function agent-
wfa-process-simple-request-answer We record the answer
message in the last-subrask-result slot. We call the dynamic part of the skill

- if the result is the keyword :abort, we kill the subtasks and clean the house by broadcasting necessary messages, we reset agent status to :idle

- otherwise, we remove the subtask from the subtask-list.

- the subtask was a broadcast or multicast and we call the function `agent-wfa-process-broadcast-answer`

  - if our policy was to take the first acceptable answer, we call an acceptability function. If OK, we do as in the simple case, canceling subtask by means of a broadcast message.

  - if we wanted to collect answers, then we save it in a special slot until some condition is satisfied (number of collected answers, or end of collecting time reached, or some other condition...) (not implemented yet)

  - we can’t recognize where the answer comes from and we declare an error.

After processing the answer we check if we have more subtasks to process. If so we wait for the next result, otherwise we exit.

The final cleaning job is done by the call to dynamic-exit in the dynamic skills. If this was not done, then we are in trouble, since no answer has been sent back to the requesting agent. The system will either hang up or send a new request and give up after a while. This can be checked: if the task-in-progress slot has not been wiped out, then the dynamic exit has not been executed. We send an error message explaining the situation. Currently we stop the process.

### 9.3 Processing an :abort Message

When an agent is receiving an abort message, then three cases can occur:

- the agent (main) is executing the task to be aborted

- the task is waiting to be executed in the agenda bin

- the task is no longer present

when the task is being executed, then we kill (and recreate) the main process. We then clean the various pieces, careful not to let the task finish so that we'd kill the wrong one.

We then send an error message to the agent that send to subtask.

### 9.4 Processing a :cancel Message

Same as :abort but we do not send an error message to the agent that sent the job to do.
9.5 Processing an :inform Message

Agent received an inform message. Look for skill associated with the :inform message, checking for possible preconditions. Apply skill. No answer is requested. The inform message is processed by the scan process. This it should not lead to long processes.

9.6 The Time Limit Mechanism

When a time-limit is associated to a request message, then a timer process is created by the agent-idle-process-request function. The process is recorded in the process-timer slot of the current task-frame.

- If the work is done before the timer triggers, i.e., we finish the job before the timer fires, then, when sending the answer, the timer will eventually fire later with no effect.

- Otherwise, when the timer fires and the task is not finished, then the timer process kills the task, does some clean up and sends an error back to the requesting agent.

More details:

The timer process is established by calling set-task-timer <agent> <task-id> <delay> where <delay> is the allowed time expressed in 1/60th seconds. The delay is computed with respect to the date (time-stamp) of the message.

The timer process waits <delay>/60 seconds and fires. At that time the function agent-process-task-time-limit is called.

- if the active task is not that specified by the timer, then the timed task has completed and we simply kill the timer process.

- otherwise, we check if a time-limit function was provided with the skills by executing the function agent-get-skill-time-limit

  . if a user function was defined, then we execute the user time-limit function

  . If not, we do nothing, unless the retry count is 0 and in that case we kill the timer and abandon the current task by calling the abandon-current-task function. The function aborts task and subtasks and sends an error message to the agent that requested the task.

If no user function was defined and the retry number is greater than 0, then the timer waits another <delay>/60 seconds and fires again.

9.7 The Timeout Mechanism

A timeout is associated to a subtask sent by an agent A to an agent B. It is processed by agent A. If an answer is not received when the timeout fires, then agent A must decide what to do. Usually it repeats the message a certain number of times before it decides to quit. The problem then is to manage the different messages that have been sent, since they all can be active when the subtask took longer than expected. On
the first received answer if there has been more than one message sent to
the agents, then all the different versions of the task request must be
canceled (broadcast).

A timeout timer is established when some agent sends a subtask using the
function \texttt{send-subtask}

A timeout is attached to a subtask and recorded in the subtask-frame of the
subtask whether the subtask is sent point-to-point, multicast or broadcast,
and whatever the protocol. The subtask frame records a pointer onto the
timer process, the timeout duration, and the repeat-count (set to 0 on
the first time). When the subtask is repeated for some reason (repeat-count
> 0) then the entries are not updated.

The timer is established by the function \texttt{set-timeout-timer} which creates a
process running the function \texttt{timeout-timer} which essentially sleeps until
the delay is exhausted.

Then

- if the timeout timer fires before we got an answer for the subtask, then
  the function \texttt{agent-process-timeout-message} agent subtask-id is
  invoked.

- . If the user has provided a timeout function to handle the case,
  then we apply the user function (defined in the skill) and if
  successful, we remove the subtask subframe from the subtask list,
  and kill the timer process.

- . otherwise (no user function, or failed user function)
  
  - if the repeat-count is greater than the max default repeat count
    \texttt{*max-repeat-count*} we abort the current task, kill and recreate
    main process, and kill the timer.

  - Otherwise, we reissue the task increasing the its repeat count,
    noting the change in the new message AND in the task
    subframe.

- otherwise it the answer to the subtask comes back before the timeout
  fires, then the answer is processed by the function \texttt{agent-waiting-for-
  answer-process-answers} that kills the corresponding timer if the
  subtask is still recorded.

9.8 Contract-net

When using the contract net, an agent sends a :request message to a set of
other agents indicating that it wants to use the contract-net protocol. Thus,
the message is a broadcast or a multicast. The execution occurs in 2 steps:

- a first step allowing to decide what agents will execute the subtask

- a second step during which results are collected
During the first attribution step, the agent requesting the task, called the manager, sends a normal message containing a description of the task to execute to the set of target agents, using the function send-subtask with a :contract-net value for the :protocol parameter. The send-message function changes the type of message to :call-for-bids, and,

- if the variable *contract-net-strategy* is set to :collect-answers it starts a timer using the function set-call-for-bids-timer

The timer will wait *default-call-for-bids-timeout-delay* 1/60 s and repeat *default-call-for-bids-repeat-count* times.

- if the variable *contract-net-strategy* is set to :take-first-answer then, the function will not set up a timer.

After sending the call-for-bids message the agent waits in the :waiting-for-answer state.

9.9 Processing a :call-for-bids Message

When an agent receives a :call-for-bids message, then its comm process calls the agent-scan-call-for-bids function.

- if the agent has not the requested skill, it simply ignores the message

- if it has a recorded answer to the task (because it already has done the same task previously, then it fetches it and sends a :bid-with-answer message

- otherwise, it checks if preconditions for the skill apply in case the user specified preconditions

- it tries to estimate when it can process the task, how long the processing will take and what will be the quality of the result, using the agent-estimate-task-parameters that returns a list of the 3 parameters

- it then inserts into its agenda a :soft-request message that will trigger an execution of the task by the main process, in case there is nothing better to do

- finally it returns a bid message to the manager

- ... keeping a copy in the pending-bids slot.

9.10 Processing a :bid Message

When an agent (in the role of manager) receives a bid, its scan process calls the agent-scan-bid function.

- if the agent has no task in progress being executed, or if there is no subtask reference corresponding to the bid (the task could have completed, be aborted, or some other agent could have provided the answer to the subtask), or if the task has already been granted (contractors slot of the subtask-frame non empty) then the bid is simply ignored.

- otherwise, we have a good bid.
. if the mode is collecting answers, then the bid is stored in the saved-bids of the subtask-frame, and the scan task resumes its watching job.

. otherwise, if the mode is :take-first-answer, then a grant message is sent to the bidder(s) and the are recorded in the contractors slot of the subtask-frame.

. otherwise an error is declared with unknown mode (this should not occur).

When the CN strategy is to take the best offer, the choice occurs when the timer fires (call-for-bids-timer). When this happens,

- first a light-blue notch is drawn on the graphics trace

- the function agent-process-bid is called. It must select the best bid, which may be complex:

  . if agent is no longer executing a main task, or if the subtask corresponding to the timer has disappeared from the subtask-list, then we quit, killing the timer

  . otherwise, we call the function select-best-bids that returns

    - nil, if no bid was agreeable

    - a list of selected bids otherwise.

    In that case

      . a cancel-grant message is broadcast

      . other pending bids of the awarded agents are upgraded regarding their possible starting time using the function agent-update-received-bids

    Then, the message is sent and the timer process is killed.

9.11 Processing a :grant or :cancel-grant Message

Then an agent having submitted a bid and being awarded the contract may receive

- a grant message, when the CN-strategy is to take the first bid

- a cancel-grant message if the CN-strategy is to take the best offer.

The agent may be :idle, executing the subtask "in case", or executing some other task "in case". If it is executing a regular task and is in the :busy state, then the message is queued.

- grant-message: the scan process executes the function agent-scan-grant recover pending bid corresponding to bid that was accepted. If lost then we print a warning and exit

  If the agent is :busy in case we call function agent-scan-busy-in-case-grant
. If the task is currently being executed, then simply changes the status from :busy-in-case to :busy

. If the task is in the agenda, with a :soft-request message then we must stop the currently running subtask and change :soft-request to :request in the message contained in the agenda. The new request message will be picked up by the main process. We reinsert the message for the task being interrupted into the agenda for a possible later "in case execution"

If the task is not in the agenda, then it has been already processed, but we do not check for that...

- cancel-grant-message: the scan process executes the function agent-scan-cancel-grant

. If the message is a cancel message, we cancel the task, removing the bid from the list of pending bids, using the function agent-scan-cancel

. If the message is a grant message, then we do as in the grant message case.

9.12 Processing a :soft-request Message

A soft-request message is processed when we are committed to a task but were not granted the contract. We set up a task, calls the skill, and, if present executes it, only if local, i.e., if it does not spawn subtasks. A return message is produced, when we finish the task. If the task is non-local then it is ignored (not undertaken).
APPENDIX

The appendices describe the structure of the different source code files and of the corresponding functions.

OMASWA code is contained in a number of different files:

2-omaswa-globals.lisp
2-omaswa-UDPnet.lisp
2-omaswa-agents.lisp
2-omaswa-messages.lisp
2-omaswa-comm.lisp
2-omaswa-self.lisp
2-omaswa-world.lisp
2-omaswa-API.lisp
2-omaswa-processes.lisp
2-omaswa-control.lisp
2-omaswa-task.lisp
2-omaswa-control-panel.lisp
2-omaswa-graphics.lisp
2-omaswa-agentwin.lisp
2-omaswa-proxywin.lisp

The content of each file is reviewed and discussed in the appendix.
Global Variables (file: globals)

This file contains all global definitions and useful functions. Most of the global variables controlling graphics are for use with the Mac window interface.

10.1 Various Windows

Several parameters keep track of some windows. The top-listener is used to get a text trace of the system behavior.

(defparameter *OMAS-window* nil "simulator control panel")

(defparameter *graphics-window* nil "graphics trace window")

(defparameter *OMAS-listener* *top-listener*)

10.2 Assistant Agents

Personal assistants interact with their master by means of a dialog. The root of the dialog must be defined.

(defparameter *initial-conversation-entry* () "conversation dialog entry for all assistant agents")

10.3 Net broadcast

Controls if we are using a distributed system or not.

(defparameter *net-broadcast* nil "indicates when the net is active")

10.4 Other Global Variables

(defparameter *agents* () "list of all locally active agents")

(defparameter *agent-list* nil "list of local agents to be displayed in the time graph")

(defparameter *broadcast-strategy* :take-first-answer "specify broadcast/multicast strategy. Can be :take-first-answer or :collect-answers")

(defparameter *default-call-for-bids-repeat-count* 3 "number of times we renew our wait when no bids present.")

(defparameter *contract-net-strategy* :take-first-answer "specify contract-net strategy. Can be :take-first-answer or :collect-answers")

(defparameter *clock* 0 "global clock") ; should be removed

; the *debugging variable is only used by OMAS but should be available centrally
(defparameter *debugging* t "Debugging flag. If set let the top listener stay on the left of the screen.")

(defparameter *drawing-timeouts* t "display timeout processes on graphics window")

(defparameter *dummy-subtask-list* () "list of dummy subtasks")

(defparameter *graphics-no-bids* nil "if t bids are not displayed on the graphics trace.")

(defparameter *last-drawn-message* () "keeps last message to avoid printing the broadcast/multicast messages several times")

(defparameter *last-printed-message* () "keeps last message to avoid printing the broadcast/multicast messages several times")

(defparameter *max-repeat-count* 2 "number of times a message will be repeated after a timeout")

;;; Warning: button colors currently inactive with Mac OS 8.6

(defparameter *OMAS-button-active-color* 16751520 "pink when active")

(defparameter *OMAS-button-inactive-color* 9618240 "light green when inactive")

(defparameter *OMAS-text-background* 16381389 "background color")

(defparameter *ontology* () "ontology for normalizing concepts: entries are a concept associated with generic concepts. Useful for normalizing skills")

(defparameter *OMAS-verbose* nil "controls the debugging trace")

(defparameter *subtask-number* 0 "generator for subtask names e.g. FAC-ST-0, FAC-ST-1,...")

(defparameter *task-number* 0 "generator for task names e.g. T-0, T-1,...")

(defparameter *trace-messages* t "flag if true messages are traced")

(defparameter *user-messages* nil "list of messages produced by the user")

10.5 Timing Defaults

(defvar *default-call-for-bids-timeout-delay* 30 "time to wait for an answer to a call-for-bids: 0.5 s")
(defparameter *highest-time-limit* 60000000 "1000000 seconds a little less than 2 yr")

10.6 Some Service Functions

(defun alt-list-set (list value prop)
  "replacing whatever was associated with prop by value in
  alternated list. ~
  Add it if prop not there."

(defun replace-or-add-pv (prop values sequence)
  "replacing whatever was associated with prop by values in
  alist ~
  Add it if prop not there."

(defun add-item-if-new (item sequence)
  "add an item at end of list if new (test is eql)"

(defun %alist-add-value (sequence prop value)
  "adding value if a new one (test is eql) to old ones in
  the alist"

(defun remove-at-pos (nn ll)
  "function that removes an item in the nth position of a
  list (non destructive)"

(defun ref (symbol)
  "traces in what function a specific symbol appears by
  examining the code of the functions in the corresponding
  file. Uses a global variable omaswa-files that contains the
  list of the omaswa files."
11  Net interface (file: netUDP)

This file contains the interface needed for running OMAS-WA distributed on several machines. It implements message exchanges using a UDP transport protocol in broadcast mode. This interface is unique to the Mac environment and uses the OpenTransport library.

A global variable *distributed-omas* 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.

Broadcasting and listening are done by omas centrally. Four functions allow to use the net services:

INITIALIZE-NET-BROADCAST broadcast-mask broadcast-distance service

This function starts or wakes up Open Transport, preparing the data structures and necessary buffers. broadcast-mask id the broadcast address of the network, e.g., "172.17.255.255" port. Broadcast distance specifies the maximum number of gates that a broadcast message can cross before dying. Default is 1 (messages restricted to the local loop. service is the name of the service (default is "omas" corresponding to port 150.

NET-SEND message

Sends a broadcast message on the web using the UDP protocol. The object message is first transformed into a string that must be shorter than the maximal allowed length. Messages are currently limited to 512 bytes.

NET-RECEIVE

Listens to the net on a separate process and extracts messages, putting them as strings in a special queue *incoming-net-messages*. Another process takes each message in turn and restructures it using read-from-string. If an error occurs the message is forgotten, otherwise the message is distributed in the mailboxes of the local agents.

TERMINATE-NET-BROADCAST

Cleans up the various structures and closes OpenTransport.

11.1 Macros and Global Variables

(defVar *err* 0 "variable for keeping track of errors")

(defparameter *opentransport-structures-active* nil "t whenever OpenTransport is activated and structures are set up")

(defparameter *big-brother* nil "listening process")

(defparameter *net-mailman* nil "dispatching process")
(defparameter *incoming-net-messages* ()
  "a queue where we save string-messages coming from the net")

(defparameter *incoming-message-stack* ()
  "stack onto which we push incoming messages")

Open transport needed structures:

(defparameter *knetbufdataisotbufferstar* #xfffffff)

(defparameter %ep% (%null-ptr) "a macptr to the endpoint structure")

(defparameter %eprcv% (%null-ptr) "a macptr to the endpoint structure")

(defvar *junk* nil "a variable that contains error results")

(defparameter %inputbuffer% (%null-ptr) "macptr to the input buffer")

(defparameter %outputbuffer% (%null-ptr) "macptr to the output buffer")

(defparameter %sndcall% (make-record :TUnitData) "parmTable for Snd")

(defparameter %rcvcall% (make-record :TUnitData) "parmTable for Rcv")

(defparameter %hostdnsaddress% (make-record :DNSAddress))

;;; create and allocate a Flag status for the Snd and Rcv functions
(defrecord flag (flags :OTFlags))

(defparameter %junkflags% (make-record flag))

(defparameter *ktransferbuffersize* 512 "size of a transfer buffer")

11.2 Service Functions

Those are very low-level functions.

(defun make-string-from-buffer (%macptr nn)
  "take nn bytes in a Mac buffer and returns a Lisp string. The functions
assumes that the codes correspond to a valid string, i.e., it does not check if
included quotes are preceded with a backslash.
Arguments:
%macptr: a macptr to the buffer")
nn: the number of chars to extract from the buffer."
defun ot-local-interface-ip-address (&optional (interface #$kDefaultInetInterface))
   "get the local IP"
defun ot-negotiate-1-byte-option ($ep% level name val)
   "Process an option (option value must be 1 byte)."
defun ot-negotiate-4-byte-option ($ep% level name val)
   "Process an option (option value must be 4 bytes)."
defun %ot-udp-release-structures ()
   "release OT structures and Mac eap stuctures"
defun %ot-udp-allocate-io-buffers (buffer-size)
   "allocate input and output buffers on the Mac heap. Names are
   %InputBuffer%  
   %OutputBuffer%
Arguments:
buffer-size: length of buffer in bytes."
defun %ot-udp-bind-endpoint (%endpoint &optional (interface-ip 0) (port 150))
   "Binds the current Endpoint to an interface IP and a corresponding port. ~
   for broadcasting, IP must be 0, port is chosen by the user.
Arguments:
   %endpoint: a macptr to the endpoint structure
   interface-IP: IP 32bit number (default 0)
   port: port number (default 150)."
defun %ot-UDP-create-endpoint (&aux %endpoint err)
   "Opens a new endpoint and returns the macptr to it."
defun %ot-UDP-send-message (%endpoint message to &aux err)
   "send a message as a UDP datagram. if to is ALL, then broadcasts the
   message. Uses %OutputBuffer% allocated in the Mac area, and %sndCall% control.
Arguments:
   endpoint: a macptr to an endpoint structure
   message : a Lisp string
   to: a target-host-name-and-port or :all (broadcast)."
defun %ot-UDP-set-synchronous-nonblocking-mode (%endpoint &aux junk)
   "set the communication mode to synchronous and blocking.
Arguments:
   %endpoint: macptr to the endpoint structure."
defun net-dispatch ()
   "Waits until messages arrive on the "incoming-message-stack". Then
   processes the last message. If it reads to a lisp expr, then distributes it into
   the input box of the local agents, otherwise simply ignore it."
defun net-message-to-string (message)
   "takes an object and builds a compact string resulting from an alternate list
   where property values are replaced by numbers and null properties are
   omitted."

defun net-string-to-message-object (message-string)
   "transform an alternate list to a structured message object.
Arguments:
   string: string representing the message
Value:
   nil if message cannot be read
   message-object otherwise."

11.3 API

Functions to be called externally.

defun net-initialize-broadcast (broadcast-address &optional
   (broadcast-distance 1) (service "omas"))
   "This function starts or wakes up Open Transport,
   preparing the data structures and necessary buffers.
   broadcast-mask id the broadcast address of the network,
   e.g., 172.17.255.255 port. Broadcast distance specifies the
   maximum number of gates that a broadcast message dan cross
   before dying. Default is 1 (messages restricted to the
   local loop. service is the name of the service (default is
   omas corresponding to port 150."

defun net-send (message)
   "Sends a broadcast message on the web using the UDP
   protocol. The object message is first transformed into a
   string that must be shorter than the maximal allowed
   length. Messages are currently limited to 512 bytes."

defun net-receive ()
   "Listens to the net on a separate process and extracts
   messages, putting them as strings in a special stack
   *incoming-message-stack*. Another process takes each
   message in turn and restructures it usinf read-from-
   string. If an error accours the message is forgotten,
   otherwise the message is distributed in the mailboxes of
   the local agents."

defun net-terminate-broadcast ()
   "empties all lists, reset global-variables, release OT
   structures and closes OT"
Agents (file: agents)

This file contains the structure and service methods for OMAS agents, as well as macros for creating agents, proxies, and skills.

OMAS means Open System of Asynchronous Agents. Basically OMAS and SMAS use agents with the same structure. SMAS is a simulation system allowing to test and debug agents in simulated time, OMAS is a multi-threaded MAS environment for executing real tasks. OMAS is somewhat a single processor version of OSACA.

An agent is a complex object structure (object composed of sub-objects). We extend accessors and methods so that they can access properties of sub-objects directly from the object.

12.1 Macros for Accessing properties

```
defmacro def-ag-set (prop sub-object-name)
defmacro def-ag-set-no-check (prop sub-object-name)
```

Example of use: (def-ag-set input-messages comm "COMM") should produce

```
(defvar output-messages (defmethod (setf output-messages) (val (self agent) )
  (let ((item (comm self)))
   (if item
     (setf (output-messages item) val)
     (error "non existing COMM part in agent ~S" (name self)))
   val))
```

```
defmacro extend-method (method subobject)
  "macro to extend the method applying to a subpart of an agent, so that it can be applied to the agent as a whole. Includes the definition for defsetf"
```

12.2 Agent Structure

```
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
  agenda ; list of messages (waiting tasks)
```
status           ; :idle or :waiting-for-result
previous-status  ; recorded previous status
status           ; task being executed
last-subtask-result ; last subtask answer message that was received
pending-tasks    ; subtask list needed when executing
                  ; the skill
pending-bids     ; active bids still pending
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              ; (?)
abilities
skills             ; list of skills (subclass)
master-bins
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              ; next thing to do
to                 ; addressee
start-time         ; start time of the task
timeout            ;
repeat-count       ; default 3
timeout-process    ; from external source
time-limit-process ; internal limit
quality            ; quality of the task
duration           ; duration of the task default 100000000
saved-answers      ; answers to a contract net
SKILL
name               ; name of the skill
description        ;
static-fcn          ; static function name
dynamic-fcn         ; dynamic function name
static-pattern      ; arg pattern
how-long-fcn       ; function to estimate time to run
how-long-left-fcn  ; function to estimate time left to run
time-limit-fcn      ;
timeout-handler     ; function called on a timeout
preconditions       ; preliminary checks

Tasks and skills are objects
12.3 Service Functions and Macros for the Agent Structure

`defmacro %agent-add (agent value prop)`

"internal macro that adds a new value to the values
associated with a property. users should use the agent-add
function.
Ex: (%agent-add mul-1 mes-3 :input-messages)."

`defun agent-add (agent value prop)`

"function using the %agent-add macro. Necessary because
the macro requires the actual name of the property to
function correctly, and cannot use a variable whose value is
this name.
Example:
  (agent-add mul-1 new-info :environment)
Arguments:
  agent: agent
  value: value to be added to the list of values
  associated with prop
  prop: keyword giving property name (will be transformed
        into the right symbol)."

`defun agent-print (agent)`

"print info about an agent. This function was used for
debugging and prints only very incomplete info about an
agent. Obsolete."

`defmacro %agent-remove-value (agent value prop)`

"macro removing a value from the list attached to a given
property. prop must be a symbol or a keyword specifying
the name of the queue, not a variable.
Ex: (%agent-remove-value mul-1 mes-3 :input-messages)."

`defun agent-remove-value (agent value prop)`

"function removing a value from the list attached to a
given property
Ex: (agent-remove-value mul-1 mes-3 queue)
Arguments:
  agent: agent
  value: value to be added to the list of values
  associated with prop
  prop: keyword giving property name (will be transformed
        into the right symbol)."

`defun agent-reset (agent &key reset-self (reset-world t)
  reset-tasks reset-skills no-position no-log)`

"reset an agent input/output trays (comm part), control
part and appearance part.
Arguments:
  agent agent to be reset
  reset-self: (key) if t reset data, goals, memory,
              intentions
  reset-world: (key) if t reset acquaintances,
environment, external-services
  reset-tasks: (key) if t reset projects, waiting-tasks
  reset-skills: (key) if t reset skills
  no-position: (key) if t does not reset agent position in
               graphics trace
  no-log: (key) if t does not reset logs (the user should
           not reset clock or task-number)."
defun agent-set-status (agent status)
  "set the status of an agent, recording the current status
  in the :previous-status slot"

defun agent-set-trace (agent)
  "set the trace flag of an agent so that each function
  applied to the agent is traced. To remove use agent-remove-
  trace. Sets the :trace flag on the p-list of the agent."

defun agent-untrace (agent)
  "reset the :trace flag to nil, stopping the trace process
  on a particular agent."

12.4 Agent Process Task

defun agent-process-task (agent &aux message)
  "the function is associated with the agenda of the agent.
  As long as there are tasks on the agenda, will pick one and
  execute it.
Arguments:
  agent: agent."

12.5 Macros for Creating Agents and Skills

defmacro defagent (name &key redefine hide learning)
  "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)
  redefine: (key) if t allows the agent to be redefined
  hide: (key) if true, hide the agent from the user
  (default is nil)
  learning: (key) if t, allows the agent to remember
  results of tasks (default ~
  is nil)."

defmacro defskill (skill-name agent-name &key static-fcn
  dynamic-fcn static-pattern how-long-fcn how-long-left-fcn
  time-limit-fcn timeout-handler preconditions)
  "macro that builds a specific skill and add it to the
  list of agent's skills
Arguments
  skill-name: name of the skill (action parameter in the
  messages
  agent-id: name of the agent receiving the skill
  static-function: (key) function implementing the static
  part of the skill
  dynamic-function: (key) function launched when a subtask
  answer comes back
  static-pattern: (key) unused
  how-long-fcn: (key) function for estimating how long a
  task should take
  how-long-left-fcn: (key) same for estimating time left
  to run the task
  time-limit-fcn: (key) function for handling time-limit
  interruptions
  timeout-handler: (key) function to process timeout
  conditions
preconditions: (key) function implementing any precondition, when it returns nil, then the skill cannot be executed."
13 Messages (file: messages)

This file contains the definitions and methods for the class MESSAGE. Messages are instances of objects. They must be flattened when exchanged via the net. As such they cannot contain sub-objects as values of their properties.

13.1 Macros and Service functions

```lisp
defmacro message-make (&rest args)
"Used for compatibility with previous code. Calls make-instance."
defun time-string (time)
  "get an integer representing universal time and extracts a string giving hour:minutes:seconds"
```

13.2 Message Objects

Messages are built from two classes: basic message and message.

**BASIC-MESSAGE**

- name: e.g., MM-23
- type: e.g., :request, :answer, :abort, ...
- date: time at which the message is profuced
- from: name of the sender
- to: name of the receiver
- action: requested action
- args: args for the requested action
- contents: e.g., answer
- error-contents: some explanations in case of error
- timeout: timeout delay
- time-limit: time allowed to execute the task
- protocol: e.g., :simple-protocol or :contract-net

:documentation "BASIC-MESSAGE is a message from which all other messages in the system are constructed. It contains only the minimally required slots."

**MESSAGE** (BASIC-MESSAGE)

- but-for: used in cancel messages
- task-id: name of the task, e.g., T-243
- reply-to: continuation
- repeat-count: number of times a message is remitted
- task-timeout: used by contract-net call-for bids
- previous-status: obsolete

13.3 Methods

The following methods allow to work with object ids while storing object names in a message. Conversion is done automatically by the method. The value attached to the property can be:
- the name of a local agent
- the keyword :all or ALL
- a symbol that will be taken as the name of an unknown remote agent.
from and to accessors are redefined and a new initialize-instance method is cooked up for correct behavior when using make-instance.

```lisp
defmethod from ((mm basic-message))
defmethod (setf from) (value (mm basic-message))
defmethod to ((mm basic-message))
defmethod (setf to) (value (mm basic-message))
defmethod initialize-instance ((mm basic-message) &key from to)
```

Other methods:

```lisp
defmethod message-clone ((mm MESSAGE))

"clone a given instance of message, using the copy-instance primitive. Must not forget to initialize the name in the copied message."

defmethod message-copy? ((ma MESSAGE) (mm MESSAGE))

"test if message ma and mm are copies."

defmethod print-object ((mm message) stream)

"redefines the printed content."
```

### 13.4 Global Variables

The following global variable is used to code messages sent over the net as strings. To save some space, message properties are coded as integers. A dictionary is used to translate from properties to numbers and vice-versa.

```lisp
defParameter *message-property-dictionary*

"something like: ((\name . 1)(\TYPE . 2) ... (previous-status . 18))"
```

### 13.5 Service Functions

In the previous OMAS and SMAS system all services were defined in terms of function. They could be defined as methods.

```lisp
defun message-build-from-view (view properties message-name)

"called to build a message from the information provided in a message window in which values may have been edited. ~

Returns the message object.

Arguments:

view: view for editing the message."

defun message-edit (message)

"opens a temporary window for editing an old message and eventually sending it.

Arguments:

message: message to be edited"
```
defun message-format (message)
  "builds a short string to display messages into agent
windows
  <date type from/to task-id action args contents>"

(defun message-get (message accessor-keyword)
  "get the value through the proper accessor for a message.
For the sake of compatibility with version 2, uses slot-
value bypassing accessors"

(defun message-limit-text-length (text limit)
  "takes a text and limits its length (used for printing)
Arguments:
  text: string to be checked
  limit: max length."

(defun message-make-abort (agent to-agent subtask-id)
  "building an error message to send back to the caller. We
could not process the job. The message will be passed to
the user with an :error content."

(defun message-make-answer (agent to-agent task-id skill
  args result &key duration quality)
  "building an answer message for a specific task.
Arguments:
  agent: agent
  to-agent: agent that will get the result
  task-id: task for which answer is provided
  skill: skill that was used
  args: args that were used
  result: result provided
  duration: (&key) time the task took to process
  quality: (&key) quality of the result."

(defun message-make-bid (agent job-parameters CFB-message)
  "building a bid message from the corresponding call-for-
bids and the parameters.
Arguments:
  agent: agent
  job-parameters: time quality accuracy
  CFB-message: message from which info is taken
    to <- from slot
    action, task-id, args from corresponding slots."

(defun message-make-cancel (agent to-agent subtask &optional
  but-for)
  "building a cancel message to send to a subcontractor. We
received an answer and cancel additional requests.
Arguments:
  agent: agent
  to-agent: to-agent
  subtask: subtask-id
  but-for: (opt) agent to which subtask will be granted."

(defun message-make-cancel-grant (agent to-agent subtask
  &optional but-for)
  "building a cancel-grant message to send to a
subcontractor. We received an answer and cancel additional
requests.
Arguments:
  agent: agent
to-agent: to-agent
subtask: subtask-id
but-for: (opt) agent to which subtask will be granted."

defun message-make-error (agent received-message &optional
(contents :error))
"building an error message to send back to the caller. We
could not process the job. The message will be passed to
the user with an :error content.
Arguments:
agent: agent
received-message: message for which error is issued."

defun message-make-grant (agent to-agent subtask)
"building a grant message to send to a list of
subcontractors.
Arguments:
agent: agent
to-agent: list of agent to which the job will be granted
subtask: subtask-id." 

defun message-make-reply (agent received-message result)
"building an answer message after the task has been
completed.
Arguments:
agent: agent
received-message: message granting the task
result: result of applying the skill (put into the
contents slot)."

defun message-make-window (&rest properties)
"opens a temporary window for defining a new message and
eventually sending it.
Arguments:
properties: (rest) if present, window will contain only
specified properties."

defun message-parse-value (prop data)
"parsing string data attached to a particular message
slot. Used when building ~
messages interactively through the message window.
Arguments:
prop: some propriety of the message
data: string obtained from the subview area."

defun message-print-slot (prop message)
"printing info into a slot of the message edit window in
such the way that it ~
can be read back.
Arguments:
prop: name of slot
message: object message." 

defun message-set (message value accessor-keyword)
"sets a value within the message by using an accessor
keyword. Must return message. Actually modifies the message
object.
Arguments:
message: message
value: value
accessor-keyword: e.g., :task-id."
defun message-net-string-to-object (message-string)
  "transform an alternate list to a structured message object.
Arguments:
  string: string representing the message
Value:
  nil if message cannot be read
  message-object otherwise."

defun message-object-to-net-string (message)
  "takes an object and builds a compact string resulting from an alternate list
where property values are replaced by numbers and null properties are
omitted."

defun message-update-from-view (view properties message)
  "called when sending message from a view in which values have been edited.
Return the message object.
Arguments:
  view: view for editing the message
  properties: list of slot names
  message-name: name of the message whose value is the message object."

;;; user related functions

defun user-receive-message (message)
  "Debugging: simply prints the message on the user channel, when the receiver is nil."
14 Communication (file: comm)

This file contains functions dealing with the communication process. In practice the functions are mainly used to analyse input messages, since all messages are directly sent by the send-message function.

In the OMAS version an agent when created (after a defagent) has two processes:

- one for running the skills that is started and put into a wait state
- another one, the scanner, for processing the input-messages

During its life more processes can be created

- for processing call-for-bids
- for processing inform messages
- for handling its master in case of an assistant agent

When a message is received, the scanner processes it. I.e., it decides

- whether it should be ignored (depending on the state of the agent),
- whether it should interrupt what the agent is doing,
- whether it should be inserted into the list of tasks to do (agenda),
- whether it should create a special process to process it.

The agent process processes task it finds in the agenda until completion, unless it is interrupted by the scanner (:abort, :cancel, :kill).

14.1 Main Agent Driver

defun agent-scan-message-for-us? (agent message)
  "checks whether the message in the box was sent to the agent, either
  - point-to-point
  - broadcast
  - multicast (agent name in the list)
  If so, accepts the message
Arguments:
  agent: agent
  message: received message to check
Value:
  message where the to field has been changed to agent if
  the message was for us, nil otherwise."

defun agent-scan-messages (agent &aux message)
  "the function is associated with the input mailbox of the agent
  and waits for incoming messages. When a message is received the scanner processes it. I.e., it decides
  - whether it should be ignored (depending on the state
  of the agent),
- whether it should interrupt what the agent is doing,
- whether it should be inserted into the list of tasks to do (agenda),
- whether it should create a special process to process it.

Arguments:
agent: agent.

\begin{verbatim}
defun agent-scan-process-message (agent message)
  "takes an input message and decide what to do with it. if the agent
  is an assistant agent, then special actions can be taken, like
  transferring the message to the user unprocessed...
  Arguments:
  agent: agent
  message: input to be dispatched."
\end{verbatim}

14.2 Functions Used by an Assistant

The following functions are used by assistant agents only. Service agents (agents providing services, but not attached to anybody) do not use such functions.

\begin{verbatim}
defun agent-assistant-abort-task (agent message)
  "message for aborting a task. If this task was issued by
  the master, then the message is saved for immediate or
  later display. Otherwise, it is processed as a regular
  abort message.
  Arguments:
  agent: agent
  message: abort message."
\end{verbatim}

\begin{verbatim}
defun agent-assistant-discard-request-message? (agent message)
  "check if there are some good reasons for discarding an
  incoming request. Currently we do not discard any message.
  Arguments:
  agent: agent
  message: incoming request."
\end{verbatim}

\begin{verbatim}
defun agent-assistant-inform (agent message &aux fn)
  "inform message. Right now we simply display it right away.
  Arguments:
  agent: agent
  message: inform message."
\end{verbatim}

\begin{verbatim}
defun agent-assistant-pick-action (agent)
  "the function is associated with the to-do bin of the
  agent and waits for data from the master. When data
  arrive in the to-do bin, then the assistant processes them, activating
  the current conversation state.
  Arguments:
  agent: agent."
\end{verbatim}

\begin{verbatim}
defun agent-assistant-process-answer-message (agent message
  &aux request)
  "answer messages are examined. If the task-id is that of
  a job sent by the master, then the answer is inserted into
  the answer-bin. Otherwise, it can be an answer to a task
  sent by the assistant, that must be in a :waiting-for-
answer state. In that case, we insert the answer into the saved-answers slot as usual. Otherwise, we discard the message.
Arguments:
agent: agent
message: request message to be processed."

defun agent-assistant-process-message (agent message)
"takes an incoming message and, depending on its type, decides whether
- to process it automatically
- to send it to the user unprocessed
- to discard it.
Arguments:
agent: agent
message: incoming message."

defun agent-assistant-process-request-message (agent message)
"request messages are examined. If the skill is present they are processed like for automatic agents, if not, their content is examined and checked according to discard reasons. If there are some discard reasons then they are discarded, otherwise, they are put into the master bin. In the last two cases (discard and transfer) the task is considered as being done.
Arguments:
agent: agent
message: request message to be processed."

14.3 Scan Functions

Those functions implement the behavior of an automatic service agent. Either they process the message immediately, or they insert it into the agenda for later processing. Immediate processing has to do with aborts, call-for-bids, or other similar messages.

defun agent-acceptable-answer? (agent message)
"right now everything is acceptable"

defun agent-scan-abort (agent message)
"function that processes a message that will interrupt the main process. This can be
:cancel (cancel a specific task)
:abort (cancel a task and send an error message to manager).
Arguments:
agent: agent
message: message to process."

When an agent is receiving an abort message, then three cases can occur:

- the agent (main) is executing the task to be aborted
- the task is waiting to be executed in the agenda bin
- the task is no longer present
When the task is being executed, then we kill the main process; we then clean the various pieces carefully no to let the task finish and kill the wrong one.

(defun agent-scan-bid (agent message)
  "agent received a bid. We first check if the task was not granted to some agents. If so, we ignore bid. Otherwise, if the contract-net strategy is :take-first-answer then we grant the task immediately canceling further offers. Otherwise, we simply add the bid to the list of saved-bids.
Arguments:
  agent: agent
  message: bid just received."

(defun agent-scan-bid-with-answer (agent message)
  "we receive a bid containing an answer, presumably from a recorded result. We simply transform the message into an answer message and put it into the agenda for the main process to examine it.
Arguments:
  agent: agent
  message: bid message containing an answer."

(defun agent-scan-call-for-bids (agent message &aux task bid job-parameters fn)
  "processes a call-for-bids in parallel with whatever the agent is doing. Checks whether the agent can and wants to answer, and if so does the following:
- check if the agent uses its memory, if so if its has already encountered the same job previously. If so, we do a table look up and send a special bid
- estimate job parameters
- commit to the job by reinserting the job into the agenda as a soft request
- return a bid to the emitter
- return earliest possible date for processing the job.
Arguments:
  agent: agent
  message: call-for-bids message."

(defun agent-scan-cancel (agent message)
  "function that processes a message that will interrupt the main process. No error message is sent back to the agent that requested the job.
Arguments:
  agent: agent
  message: cancel message to process."

When an agent is receiving a cancel message, then three cases can occur:

- the agent (main) is executing the task to be aborted
- the task is waiting to be executed in the agenda bin
- the task is no longer present

When the task is being executed, then we kill the main process we then clean the various pieces careful no to let the task finish and kill the wrong one.
defun agent-scan-cancel-grant (agent message)
  "a cancel grant message has been received. If the message
  is a cancel, then cancel the specified task. If the agent
  is in the :but-for list, then the task is granted.
  Arguments:
    agent: agent
    message: cancel-grant-message."

defun agent-scan-grant (agent message)
  "a cancel grant message has been received.
  Arguments:
    agent: agent
    message: grant-message."

defun agent-scan-inform (agent message &aux fn)
  "agent received an inform message. Look for skill
  associated with the :inform message, checking for possible
  preconditions. Apply skill. No answer is requested.
  Arguments:
    agent: agent
    message: inform message."

defun agent-make-process-name (agent mode-string &key
  fixed-name)
  "produces a string for naming processes. E.g., FACTORIAL-
  call-for-bids-22
  Arguments:
    agent: agent
    mode-string: string identifying type of job (e.g.,
      \"inform\")
    fixed-name: (key) if T, there is no version to the
      name."

defun agent-insert-into-agenda (agent message &aux task)
  "function that inserts message into the agenda to be
  processed by main agent process.
  Arguments:
    agent: agent
    message: message to be processed."
15 Representation of Self (file: self)

This file contains the definitions and methods associated with the class SELF models the self part of a given agent. SELF is used to model the agent itself. Thus, it should contain a summarized description of its skills, a description of its capabilities (features), a model of its goals and intentions, and its memory. Until now, only a rough model of the memory is implemented in a very crude fashion.

15.1 SELF Structure of an Agent

```
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
```

15.2 Memory and Memory Items

Memory is used very crudely, by recording a task and its result, for agents capable of remembering. Thus, whenever the exact same task is requested at a later stage, the agent is capable of sending the result immediately. Of course, we assume that the tasks are stable.

The agent memory contains a list of memory items. A memory item is a piece of data that is recorded by means of a tag.

```
Memory-item
   date:      time at which it was recorded
   tag:       recording key
   service:   type of service in which it was produced
   info-type: type of information (e.g., mailing-list)
   ontology:  ontology in which data is defined
   value:     data themselves
```

Most of the information can be obtained from the context object when the data is recorded.

15.3 Memory methods and functions

```
defmethod print-object ((mm memory-item) stream)
   (defun time-full-string (time)
      "produces a string with full time, e.g., \"11:15:49 2/12/2000\"."
   
   The forget function is available from the API to wipe out memory
   
   defun agent-memory-retrieve (agent input)
   "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
   input: pattern for recovering data."
```
defun agent-memory-remember (agent key value &optional service info-type ontology)
   "saves data as an object in memory an a-list. The key is memorized in the tag slot.
Arguments:
agent: agent
key: pattern for recovering data
value: data to be saved
service: (opt) service that produced the value
info-type: type of information
ontology: ontology for which the terms have a meaning."
16 Representation of the World (file: world)

Currently the representation of the world is limited to the knowledge of other agents. Such knowledge is built dynamically by noting the name and skills of an agent that was not known previously. Thus, whenever any new agent joins the group, information is gathered through the messages that each agent sees.

16.1 Structure of the World Representation

The world part of an agent is structured as:

```
world
  acquaintances  other known agents
  environment    obsolete: replaced by data in self
  external-services  knowledge of other agents
```

16.2 Acquaintances

Each acquaintance is represented by an instance of the class AGENT-INFO:

```
AGENT-INFO ()
agent
skills-requested
skills-displayed
date-of-first-
:documentation "entry describing another agent. It contains the agent name the skills we could get from the messages (either requested or displayed, and the time the agent was detected for the first time."
```

16.3 Methods and Functions for Acquaintances

```
defmethod print-object ((mm agent-info) stream)
defun agent-parse-message-for-acquaintances (agent message)
  "function called when an agent starts processing a message. Checks if the agent sending the message is known, if not, then records its name and current date, otherwise, check the skill and record it if not already known. For each agent we record the skill in one of two lists: skill displayed, or skill requested, according to the type of message. If the skill is a new information, we record it in the external-services list. Arguments:
  agent: agent
  message: message to be processed."

defun agent-reset-acquaintances (agent)
  "remove the description of other agents from the acquaintances slot of the world model of an agent. Also resets external services. Arguments:
  agent: agent."
```
defun agent-update-external-services (agent message)
  "record the skill into the list of external services. An entry on
  the list of external services looks like (<skill> <agent1> ... <agentn>).
Arguments:
  agent: agent
  message: message containing the skill to be recorded."
17  Representation of Tasks (file: tasks)

The principle of recording previous tasks is simple:

- when a task is received, then a description of the task is temporarily kept in the task description part of the agent

- when the task is completed and an answer is returned to somebody, then the description of the task is kept in memory with its main characteristics

Thereafter:

- if a task is requested with the same skill and parameters, then the result is looked up in the agent memory and the answer returned immediately

- if the agent receives a call-for-bids for the same task, then it will submit a bid-with-answer containing the result of the processing of the task. The caller will then have the choice of canceling the bidding process right away or waiting until enough answers are obtained.

When request or CFB are received for task that are already recorded, we simply record the information related to the instance of the new job (for statistics purposes).

17.1  Task Structure

A task is implemented in 2 parts: a generic one (TASK) and another one containing details (TASK-INSTANCE). When the task is saved the generic part becomes a common part to all instances of the task kept in the task-instance list.

TASK  ()

agent
skill
arg-list
answer
duration
quality
competitors
task-details
task-instances

:documentation "a task is a model of task to be recorded in the task part of the agent while the task is being processed. After the task is finished then it is cleaned and transferred into the agent memory."

TASK-INSTANCE  ()

task
task-message
task-id
start-date
customer
time-limit
protocol
continuation
grant-date
bids

:documentation "describe an instance of task corresponding to a skill and specific arguments."

17.2 Task Methods and Functions

defun agent-check-memory-for-same-task (agent message)
   "checks if the task described in the message has already been executed. If so, returns a pointer onto the task (I.e., the task object).
Arguments:
   agent: agent
   message: message containing task to check."

defun agent-record-starting-task (agent message)
   "record a task, if not already recorded in memory. Create it, fill the info from the input message and record it in the waiting-tasks list."

defun agent-remember-task (agent message &aux task task-instance)
   "remember a task, moving it from the waiting-tasks to the memory, recording the result and its quality, and the time it took to complete it.
Since there is a single task in the waiting-tasks list the following is quite simple.
Arguments:
   agent: agent
   message: message specifying the task."

defun agent-update-task (agent message)
   "we have a task entry in memory containing info for the task being processed by the agent, and we want to update the entry with new results. This should not happen if the agent had recognized that the task to do was the same as one it has done previously. Thus we do nothing and send a warning.
Arguments:
   agent: agent
   message: message containing answer to the task."
18 Representation of the Master

Not clear yet.
19 Agent Control (file: control)

Agent Control contains the library for processing tasks.

19.1 Task Structures

A task is represented in the control area as a structure called a task-frame.

**TASK-FRAME**

- **id**: id of task
- **message**: processed message by the task (message that triggered the task)
- **from**: agent that sent the message
- **reply-to**: agents that should receive the results
- **answer-being-processed?**:?
- **result**: result of the task (initially nil)
- **subtask-list**: list of task-frames corresponding to spawned subtasks
- **start-time**: start time of the task
- **estimated-length**: estimated time the task needs to be executed (default is 100000000)
- **estimated-end-of-task**: estimated end-time for the task (universal time)
- **quality**: expected quality for the result
- **time-limit**: maximum allowed time to process the task
- **timer-process**: timer for implementing the time-limit
- **environment**: area left to the user to structure intermediate data

Subtasks are recorded as subtask-frames on the subtask-list of a task-frame.

**SUBTASK-FRAME**

- **id**: id of task
- **message**: message sent by the task to the possible contractors
- **to**: targeted contractors
- **result**: result of the task (initially nil)
- **start-time**: start time of the task
- **estimated-end-time**: estimated end-time for the task (universal time)
- **quality**: expected quality for the result
- **saved-bids**: bids that were received on this subtask (contract net)
- **contractors**: agent(s) that were chosen to execute the task
- **environment**: area left to the user to structure intermediate data

We also redefine print methods for TASK-FRAME and SUBTASK-FRAME.

19.2 Main Driver

```lisp
(defun agent-process-agenda-task (agent message)
  "agent was idle and found something to do (:request, :soft-request, :internal,:answer). It calls the proper skill, and applies it, setting the status to :busy or :busy-in-case.
 "
)```
Arguments:
  agent: agent
  message: message to process.

19.3 Service Functions

defun agent-delete-pending-bid (agent task-id)
  "remove bid from the list of pending bids using the task-id.
Arguments:
  agent: agent
  task-id: task-id of the bid to be removed."

defun agent-estimate-task-duration (agent skill args)
  "for a given skill with particular args estimate how long
  the task will take.
Arguments:
  agent: agent
  skill: skill being checked
  args: whatever args are being necessary for the how-lon-fcn."

defun agent-estimate-task-parameters (agent message)
  "estimate earliest execution time, delay and quality of
  result. Take into account currently executing task, granted
  subtasks through :grant or :cancel-grant messages.
Arguments:
  agent: agent
  message: message containing the task to execute.
Returned value:
  (earliest-execution-time delay quality)"

Accessors

The following function are accessors they are kept for compatibility with
older versions of OMAS.

defun agent-get-dynamic-skill (agent action)
  "get the name of the function corresponding to the
  requested skill"

defun agent-get-how-long-fcn (agent action)
  "get the name of the function estimating processing time
  for an action"

defun agent-get-how-long-left-fcn (agent action)
  "get the name of the function estimating how much time to
  end of task"

defun agent-get-pattern-skill (agent action)
  "get the type pattern for the function corresponding to
  the requested skill. If nil then no check on arguments. Ex:
  ((x int)(y string))"

defun agent-get-static-skill (agent action)
  "get the name of the function corresponding to the
  requested skill"

defun agent-get-skill (agent action)
  "get the skill object from its name"
defun agent-get-skill-preconditions (agent skill)
  "get the name of a function for processing timeout conditions on the skill.
Arguments
  agent: current agent
  action: skill name (current task) for which we want to process the timeout."

defun agent-get-skill-time-limit (agent action)
  "get the name of the function to process timeout messages"

defun agent-get-skill-timeout (agent skill)
  "get the name of a function for processing timeout conditions on the skill.
Arguments
  agent: current agent
  action: skill name (current task) for which we want to process the timeout."

defun agent-has-skill? (agent skill)
  "check if agent has the required skill"

Functions for processing messages

The following functions constitute the engine that drives the agent. They process a message according to its type.

defun agent-idle-process-internal (agent message skill)
  "processes an internal message using adequate skills. All task set up is done by the skills themselves (system skills).
  No return message is produced.
Arguments:
  agent: agent
  message: input message (task requires answer)
  skill: requested static skill."

defun agent-idle-process-request (agent message skill)
  "set up a task, calls the skill, and, if present executes it. A return message is produced.
Arguments:
  agent: agent
  message: input message (task requires answer)
  skill: requested static skill."

defun agent-idle-process-soft-request (agent message skill)
  "set up a task, calls the skill, and, if present executes it, only if local, i.e., if it does not spawn subtasks. A return message is produced, when we finish the task. If the task is non local then it is ignored (not undertaken).
Arguments:
  agent: agent
  message: input message (task requires answer)
  skill: requested static skill."

defun agent-process-bids (agent subtask-id)
  "function called by a call-for-bids-timeout process to process eventual bids. If no bids are found acceptable, it returns nil. Otherwise, send a grant message to the sender of a winning bid."
Arguments:
agent: agent
subtask-id: subtask attached to this process."

defun agent-update-received-bids (agent to-agent bid)
  "when an agent A grants a contract to an agent B, then it
can update all the bids submitted by the agent B.
Arguments:
agent: agent
to-agent: agent that is granted the bid
bid: bid that was selected corresponding to the granted
contract."

defun agent-waiting-for-answer-process-answers (agent
current-task fn)
  "the agent has spawned subtasks and we must process
answers when they come. A waiting loop is set on saved-
answers, and each time an answer is processed the subtask
is removed from the current-task task-frame. When all
subtasks have been processed, then we send an answer back
to the agents expecting it.
Arguments:
agent: agent
current-task: task-frame for current-task
fn: function name of dynamic skill to be applied to each
answer."

defun agent-wfa-process-broadcast-answer (agent message
current-task fn)
  "here we got an answer to a broadcast message. If the
strategy is :take-first-answer then we are in the same case
as in the simple request, except that we must send a cancel
message to cancel the other jobs. If we are in the strategy
to :collect-answers then we save answers until some
condition is met. At that time we will process whatever we
collected.
Arguments:
agent: agent
message: answer message
current-task: main agent task
fn: dynamic skill of the agent."

defun agent-wfa-process-simple-request-answer (message
agent current-task fn)
  "process an answer message for a subtask. This is a
simple request.
Arguments:
agent: agent
message: answer message
current-task: task being executed by the agent
fn: dynamic skill function for the message."

defun agent-make-task-frame (agent message)
  "build a task-frame description for the task to be
executed, from the information found in the incoming
message.
Arguments:
agent: agent
message: message that creates the task."

defun agent-remove-skill (agent skill-name)
"removes the entry corresponding to the skill from the list of skills.
Arguments:
   agent: agent
   skill-name: name of the skill that should be removed from the list of skills."

defun agent-replace-task-object-in-list (task-info task-list)
   "used by agent-replace-task-info to replace an entry on the task-info list.
Arguments:
   task-info: new entry
   task-list: list of task-info entries."

defun agent-select-agenda-message (agent)
   "should allow to implement fancy algorithms to select messages in the agenda.
Currently selects the oldest from the list."

defun agent-select-answer-message (agent)
   "currently selects the oldest message in the bin."
Processes (file: processes)

Functions dealing with processes and timers.

An agent has several processes. In addition each timer is a new process.

defun noop ()

defun agent-process-timeout-message (agent subtask-id)
   "process a timeout message.
   For point-to-point mode, reemits message until *max-repeat-count* is reached in which case it sends back an :answer message to the agent with contents:timeout-error.
   Arguments:
   agent: agent
   message: timeout message to be processed."

defun call-for-bids-timer (agent subtask-id delay &optional (retry 0))
   "set up a timer to process call-for-bids timeout. The idea is to wait some time to allow bids to come in.
   - If we have some bids, then we process them to find the best one. If we have a satisfactory answer we send a cancel-grant message.
   - If none is suitable, then we try to wait some more time (until retry gets to 0).
   - If no bid is suitable and retry is 0, we have a hard timeout, and we must cancel the task and subtasks, and send back an error message to the caller.
   Arguments:
   agent: agent
   subtask-id: id of task to be watched
   delay: time to wait (in 1/60 second)
   retry: number of retries to do (default none)."

defun kill-and-recreate-main (agent)
   "kill and recreate an agent main process. Used to stop whatever the agent was doing.
   Arguments:
   agent: agent."

defun set-call-for-bids-timer (agent subtask-id delay &optional (retry 0))
   "set up a timer to process returned bids.
   Arguments:
   agent: agent
   subtask-id: id of task to be watched
   delay: time to wait (in 1/60 second)
   retry: number of retries to do (default none)."

defun set-time-limit-timer (agent task-id delay &optional (retry 0))
   "set up a timer for imposing a time-limit on task.
   Arguments:
   agent: agent
   task-id: id of task to time out
   delay: delay (in 1/60 second)
   retry: (opt) number of retries (default is 0)."
defun set-timeout-timer (agent subtask-id delay)
  "set up a timer to process a subtask timeout.
  Arguments:
    agent: agent
    subtask-id: id of task to be watched
    delay: time to wait (in 1/60 second)
  Returned value:
    process object."

defun time-limit-timer (agent task-id delay &optional
  (retry 0))
  "set up a timer to process task time-limit.
  Arguments:
    agent: agent
    task-id: id of task to be watched
    delay: time to wait (in 1/60 second)
    retry: number of retries to do (default none)."

defun timeout-timer (agent subtask-id delay)
  "set up a timer to process subtask timeout. Calls agent-
  process-timeout-timer.
  Arguments:
    agent: agent
    task-id: id of task to be watched
    delay: time to wait (in 1/60 second)."

defun agent-process-task-time-limit (agent task-id retry)
  "see whether a time-limit function is provided with the
  skills. If not, does nothing, if the retry count is 0 kills
  the timer. The task being watched should be the current
  task. If not, then we assume the task has completed and we
  kill the timer.
  Arguments:
    agent: agent
    task-id: task to watch
    retry: number of retries left."
21 A.P.I. (file: API)

The API functions have been described in Section 7 (The Agent Language). The current section reflects the content of the corresponding file.

21.1 Macros

defmacro %remove-message-from-queue (property value queue)
    macro to simplify the writing when removing messages from queues. queue must be explicit (keyword)

21.2 User and Scheduler Functions

defun abandon-current-task (agent &optional message &aux initial-message)
    "When in the process of executing a particular task, the agent might decide to abandon the task, killing all subtasks and returning to the :idle state. This is useful when an agent has a particular skill but does not want to answer the request.
Arguments:
    agent: agent
    message: message corresponding to the task to be abandoned."

defun answering-agent (agent)
    "provides the name of the agent that sent an answer. Useful to know what agent answered, e.g., in the dynamic part of a skill.
Arguments:
    agent: agent."

defun assistant? (agent)
    "check whether an agent is an assistant, in case it returns t, or not.
Arguments:
    agent: agent."

defun cancel-all-subtasks (agent &optional current-task)
    "cancel all subtasks launched by a particular agent. Get them from the list of pending tasks contained in the task-frame of task-in-progress. Does not cancel the main task in progress.
Arguments:
    agent: agent."

defun cancel-current-task (agent &key message no-subtasks)
    "same function as abandon-current-task, but does not send error message to the agent that required the task.
If no task is being currently executed, then does nothing. Otherwise the calling function should make sure that task-in-progress is the same the one that was executing when the function was called; i.e., should call without-interrupts.
We do the following
- cancel subtasks unless no-subtasks flag is set
- remove the pending-bids entries created by the task (shouldn't we cancel the bids?)"
- reset the main process if we are not in it (e.g., canceling from a timer)
- remove possible repeated requests from the agenda
- remove the waiting-task description entry if the agent is learning
- reset the task-in-progress entry (remove the message)
- reset the status to :idle (displaying the result)

Arguments:
agent: agent
message: (key) currently ignored
no-subtaks: (key) if t, indicate that no sbtasks have been spawned and that if is not necessary to send cancel messages.

(defun cancel-answering-subtask (agent)
  "cancel subtask corresponding to the received answer message being processed.
Arguments:
agent: agent."

(defun cancel-subtask (agent subtask-id)
  "Cancel the subtask corresponding to subtask-id. Any info put into the user-managed environment area should be cleaned by the user. We clean input-messages, agenda, 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.
Arguments:
agent: agent
subtask-id: id of the task to be cancelled."

(defun create-subtask-id (agent)
  "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.
Arguments:
agent: agent."

(defun create-task-id (agent)
  "Simply provides a task id. This version uses a global counter rather than gentemp so that we can reset task ids when debugging.
Arguments:
agent: agent."

(defun dynamic-exit (agent result)
  "user-called function that takes the result from a task, builds up a message to forward the answer as required. The message is build, put into the:delayed-output slot; the status is changed to :busy.
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.
Whenever dynamic-exit is called by a process that is not the main agent process then it kills and recreate the main agent process to get rid of possible waits.
Arguments:
agent: agent"
result: result to send back to the caller or to the continuation.

(defun forget (agent)
  "used to wipe-out agent's memory
  Arguments:
    agent: agent."

(defun get-environment (agent)
  "get the environment area contained in the task frame representing the current ~
  task. If no task is present declares an error.
  Arguments:
    agent: agent."

(defun get-internal-time-string ()
  "produces a string with hr:min:sec from internal time"

(defun get-time-limit (agent)
  "obtain the time-limit associated with executing task. If none, then returns most positive integer in the system.
  Arguments:
    agent: agent."

(defun master-task? (agent task-id)
  "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."

(defun omas ()
  "start function. Reset all agents, creates a controller pannel."

(defun pending-subtasks? (agent)
  "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.
  Arguments:
    agent: agent."

(defun print-everything ()
  "print the whole set of agents"

(defun purely-local-skill? (agent skill)
  "checks if a skill operates locally, i.e., not spawning 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."

(defun reset-all-agents ()
  "clear all agents, resetting input and output trays, and emptying tasks; also resetting number of tasks to 0"

(defun select-best-bids (agent bid-list)
  "select best bids in a list of bids according to job-parameters found in :contents."
For now best bid is the earlier job.
Arguments:
agent: agent
bid-list: list of submitted bids (message objects)."

defun %send-message (message)
"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 the *net-broadcast* is on, we also broadcast on the
net
Argument:
message: message to send (unchecked)."

defun send-message (message &aux target new-message)
"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
- if the target is :all then we send the message to all
agents known by the system
- if the target is :all-and-me we do the same as for
:all including oneself
- if the target is a list of agents, then we send to
each agent of the list
- if the target is a single agent, then we sent 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.
Arguments:
agent: agent
message: message to send."
task-id: (key) specific id to identify the task. It should be unique.
    by default it is created by SMAS

timeout: (key) timeout delay allowed for executing the subtask default is none

protocol: (key) protocol for the message, default is :basic-protocol.

(defun send-inform (agent &key (action 'INFORM) to args (delay 0))
    "prepares a message containing the parameters for issuing an inform to another agent. Default action is INFORM.
Arguments:
    agent: agent
    action: (key) skill to be invoked (default: INFORM)
    to: (key) receiver
    args: (key) arguments to the inform message
    delay: additional delay (default 0)."

(defun static-exit (agent arg)
    "amounts to a noop so far"

(defun slow (delay)
    "put the process in a wait state during delay time.
Arguments:
    delay: time to wait in 1/60 second."

(defun agent-trace (agent text &rest args)
    "function used to trace agent's behavior.
Arguments:
    agent: agent
    text: text for string format
    args: arguments for the format variables."

(defun trace-agent (agent)
    "put a property trace on the plist of the name of the agent.
Arguments:
    agent: agent to trace."

(defun untrace-agent (agent)
    "removes the mark allowing the system to trace agents.
Arguments:
    agent: agent to untrace."

(defun update-environment (agent env)
    "replace the agent's environment with env.
Arguments:
    agent: agent
    env: environment part of the agent."
22 Control Panel (file: control-panel)

Functions to display and obey commands given at the control panel. The functions depend on the Lisp environment being executed. The following functions correspond to MCL 4.2 in a Macintosh environment (OS 8.6). They require Quickdraw.

The screen set up is organized as follows:

```
+----------------------------------+-----------------------------+
|       OMAS    control panel      | graphics trace              |
|           panel                  |                             |
|        Agent 1                   |                            |
|                                |       Listener              |
|        Agent n-1                 |    (message trace)          |
|                                |                            |
|        Agent n                   |                            |
+----------------------------------+-----------------------------+
```

Everything is done through the control panel which is launched at the beginning. The area agent 1 to agent n are slots for displaying the content of agents specified by double clicking in the list of agents in the control panel. The system allocates a free slot in the agent area.

The graphics trace displays a diagram of what is going on.

The Listener area can be used as a Lisp listener or is used to display messages and messages when traced.

To display a control panel, execute the following:

```
(OMAS)
```

This should display a green control panel with a number of buttons.

To display an agent window, execute the following:

```
(make-agent-window 'proxy 1)
```

```
(make-agent-window 'factorial 2)
```

The OMAS control panel is shown on Figure 3.1.
22.1 Classes and Methods

OMAS-BUTTON (button-dialog-item)

Also create a class for displaying lists of values

AREA-FOR-VALUES (sequence-dialog-item)

original-list    list to be displayed in the table area

We define a control panel for OMAS:

OMAS-CONTROL-PANNEL (dialog)

input-message slot to contain a message

Now a subclass for handling input text:

OMAS-DIALOG-ITEM (EDITABLE-TEXT-DIALOG-ITEM)

defmethod view-mouse-enter-event-handler ((button OMAS-button))
  "The handler is run every time the cursor enters the button area"

defun set-button (button)
  "Set the color of a button as defined above"

;;; same as above, but disable it

defmethod view-mouse-leave-event-handler ((button OMAS-button))

defun reset-button (button)
  "Set the color of a button as defined above"

22.2 Various Functions

defun make-OMAS-control-pannel ()
  "Creates a control panel for the OMAS simulator
  The resulting window is attached to the *OMAS-window* variable."

defun OMAS-post-message (message)
  "When the control panel is shown displays the message."
Argument:
  message: message to be displayed."

defun reset-control-pannel (item)
  "redraws the control panel according to values of the
  scheduler parameter"

defun select-agents (item)
  "function for selecting agents to display in the graphics
  trace window. Agents are selected in the scroll down list
  by using C-click, then clicking the pick selection button."

defun select-message (item)
  "function for selecting a single message to edit and then
  send.
  Arguments:
    item: button being clicked."

(defun display-selection-list (item selection-list)
  "displays the list of agents or scenarios in the
  selection window, resetting the cells to unselected"
23 Agent Windows (file: agentwin)

Functions to display information about a specific local agent. The functions depend on the Lisp environment being executed. The following functions correspond to MCL 4.2 in a Macintosh environment (OS 8.6). They require Quickdraw.

Agent windows are small windows that are stacked on the left hand-side of the screen. There is room for six windows to be visible simultaneously. Two such windows are shown on Figure 23.1.

![Agent Windows](image)

Figure 23.1 - Agent Windows

23.1 Global Variables

```lisp
"We can display at most 6 agent windows simultaneously"

defparameter *agent-window-v-position*
```

23.2 Classes and Methods

```lisp
OMAS-AGENT (dialog)
  rank position of the window in the stack
  agent agent being displayed

defmethod view-click-event-handler ((ww OMAS-agent) where)
  "when double-clicking onto an agent window, show the agent's internals"
```
### Functions

```lisp
(defun agent-display (agent)
  "update agent window by filling the various fields."
)

(defun display-agent (agent &aux pos win)
  "try to display an agent. Check *OMAS-agent-window-slots* for an empty slot. if none, then we cannot display the agent. The user must first make some room. Otherwise, we take the first available space to display the agent."
)

(defun make-agent-window (agent position &aux win)
  "Creates a window for displaying an agent"
)

(defun status-color (agent)
  "pick up color according to status"
)

(defun status-label (agent)
  "pick up short label according to status"
)"
24 Graphics Trace (file: graphics)

Functions to display messages exchanged by agents. The functions depend on the Lisp environment being executed. The following functions correspond to MCL 4.2 in a Macintosh environment (OS 8.6). They require Quickdraw.

Exchanged messages are displayed with colors depending on their type ( ).

![Figure 24.1 - Graphics trace](image)

24.1 Classes and Methods

The displaying window is augmented with scrolling capabilities.

```lisp
(defmethod initialize-instance ((self v-scrolling-window) &rest rest &key (scroller-class 'scroller) track-thumb-p field-size)
  (superinitialize-instance self :window rest)
)

defmethod set-view-size ((self v-scrolling-window) h &optional v)
```

Jean-Paul A BARTHES

Printed 01/11/02
defmethod window-zoom-event-handler ((self v-scrolling-window) message)

scrolling-window-scroller (scroller) ()

defmethod view-draw-contents ((self scrolling-window-scroller))

defun repaint (self)

GRAPHICS-WINDOW (v-scrolling-window)
  last-draw-time
  last-draw-position
  clock
  count-for-names

  :documentation "graphics window in which we draw arrows corresponding to messages exchanged among agents. It records the last time something was drawn on the pane in order to compute space between two successive messages."

defun make-graphics-window ()
  "create a window in which we draw arrows corresponding to messages exchanged among agents.
  Arguments: none."

defun adjust-clock-for-drawing (&key no-minimum &aux (delta 0))
  "computes the integer value of the drawing clock in order to space the arrows accordingly. Anything less than 1 second gets a time value of 1, after 5 seconds the value is set to 5, in between values are rounded to the next integer. The time difference is computed with respect to the last time-value that was stored in the last-draw-time of the graphics window slot.
  Time is kept in the clock slot of the graphics window.
  Arguments:
    no-minimum: (key) if t the returned adjustment can be 0."

defun arrow (ww start end color &optional (size '#@(3 3)))
  "draw an arrow from position start to end of color color with optional thickness"

defun draw-line (ww start end color &optional (size '#@(3 3)))
  "draw a line from start to end of color color with optional thickness"

defun agent-display-time-limit (agent &optional (color *red-color*))
  "draw a horizontal red notch"

defun agent-display-timeout (agent &optional (color *orange-color*))
  "draw a horizontal orange notch"

defun agent-display-square (agent &optional (color *orange-color*))
  "draw a small horizontal colored notch. Used to indicate broadcasts."
Arguments:
  agent: agent
color: (opt) drawing color (default orange)."

(defun agent-draw-state (agent old-clock clock)
  "draw a vertical color bar to indicate the agent state
  thin blue is :idle
  thick red is :busy
  thick yellow is :waiting-for-answer
  thick orange is :aborting
Arguments:
  agent: agent
  old-clock: previous value of the pseudo-clock
  clock: current value of the pseudo clock."

(defun agent-draw-time-limit (agent old-clock clock)
  "draw a thin vertical green color bar to show agent time-
  limit process.
Arguments:
  agent: agent
  old-clock: previous value of the pseudo-clock
  clock: current value of the pseudo clock."

(defun agent-draw-timeouts (agent old-clock clock)
  "draw a thin vertical orange color bar to show agent
  timeout processes.
Arguments:
  agent: agent
  old-clock: previous value of the pseudo-clock
  clock: current value of the pseudo clock."

(defun state-draw ()
  "draw lines to indicate the state of agents. The function
  is called whenever the status of a particular agent
  changes, just before it changes.
Arguments:
  none."

(defun message-draw (message)
  "draw a message from an agent to another one as a color
  arrow
  blue :request
  green :answer
  red :error
Arguments:
  message: message to be drawn."

(defun message-trace (message)
  "prints the message content using the proper color unless
  it is a bid message and it was decided not to show them.
Arguments:
  message: message whose text must be displayed."

(defun scheduler-check-window-size ()
  "checks if the next cycle is going to fit in the graphics
  window. If not, then scrolls the window up some.
Arguments: none."

(defun scheduler-print-agent-names ()
  "prints the agent names into the header of the graphics
  window.
defun scheduler-set-agent-display-position ()
  "for each agent of the display list compute its horizontal position in window.
Arguments: none."
Proxy Windows (file: proxywin)

Functions to display exchanges with the master (only used by assistant agents). The functions depend on the Lisp environment being executed. The following functions correspond to MCL 4.2 in a Macintosh environment (OS 8.6). They require Quickdraw.

The implementation of the dialogs between the assistant and its master is experimental and incomplete. It will require major changes in the agent structure. E.g., the creation of various sub-objects: one for recording ontologies, one for recording master’s preferences, etc. Currently the interface is done by means of dialogs using the MOSS representation language.

25.1 Structure and Methods

OMAS-PROXY (dialog)
  rank (unused)
  agent

  :documentation "Display an agent in a specific slot (rank) below control panel"

defmethod view-key-event-handler ((item OMAS-DIALOG-ITEM) char)
  "Redefine a special key handler that tests each character until a linefeed return is encountered, in which case we transfer content of buffer to the to-do buffer of the agent bins."

OMAS-ASSISTANT-PANEL (dialog)
  agent :accessor agent)

  :documentation "Defines the format and behavior of the OMAS ASSISTANT PANEL"

25.2 Functions

defun display-proxy-window (agent &aux win)
  "we build and display a window for the dialog with the master.
  Arguments:
    agent: agent."

defun make-master-window (agent)
  "define the master's interface window, capable of handling dialogs and viewing various types of info like the contents of the master's bins."

defun proxy-activate-answer-panel (agent)
  "activates the master's panel"

defun proxy-answer-to-request (agent &key to action answer (protocol :basic-protocol))
  "after having processed the text returned by its master, the assistant uses this function to return the final answer back to the agent that issued the request, unless the to argument is specified. The protocol can also be specified."
The function uses the message recorded at the :task-in-progress level to obtain all missing information.
Arguments:
agent: agent
to: (key) receiver
action: (key) skill
answer: (key) processed text
protocol: (key) protocol to use."

defun proxy-answer-summary (message)
  "take an answer message and builds up a string that
summarizes the message content.
Arguments:
  message: message to display."

defun proxy-clear-master-input (agent)
  "clear master's input area.
Arguments:
  agent: agent."

defun proxy-detail-answer (&rest ll)
  "shows the details of an answer."

defun proxy-detail-discarded-message (&rest ll)
  "shows the details of a discarded message."

defun proxy-detail-pending-request (&rest ll)
  "shows the details of a pending request."

defun proxy-detail-waiting-message (&rest ll)
  "shows the details of a waiting message (task to
execute)."

defun proxy-discard-answer (&rest ll)
  "eliminates a specific answer."

defun proxy-display-discarded-messages (agent)
  "displays discarded messages in the corresponding pane of
the proxy com window. the count in special count window.
Arguments:
  agent: agent."

defun proxy-display-pending-requests (agent)
  "displays pending requests in the corresponding pane of
the proxy com window. upgrades the count in special count
window.
Arguments:
  agent: agent."

defun proxy-display-request-results (agent message)
  "displays the results from an answer message. In order to
do a good job of presentation, the assistant should know
the nature of the answer, which may not be the case.
Arguments:
  agent: agent
  message: message containing the answer."

defun proxy-display-text (agent text &optional (view-name
  'output))
  "takes a string and displays it in the display pane of
the proxy com window."
Arguments:
agt: a g t
answer: string to be displayed in the com window
view-name: (opt) name of the destination view (default is output).

(defun proxy-display-unprocessed-answers (agent)
  "opens a temporary window containing the list of unprocessed answers. It is then possible to process answers, and return to the dialog.
Arguments:
agent: agent."

defun proxy-display-waiting-answers (agent)
  "displays waiting answers in the corresponding pane of the proxy com window. upgrades the count in special count window.
Arguments:
agent: agent."

defun proxy-display-waiting-messages (agent)
  "displays waiting messages in the corresponding pane of the proxy com window. upgrades the count in special count window.
Arguments:
agent: agent."

defun proxy-get-canonical-entry (agent xx ontology)
  "examines if xx is a word corresponding to an entry of the ontology.
Arguments:
agent: agent
XX: entry to normalize
ontology: alist representing ontologies."

defun proxy-get-string (agent text)
  "opens a temporary window for getting text from user"

defun proxy-get-user-input (agent text)
  "get input from user transforming it into a list.
Arguments:
agent: agent
text: text to be printed in the input box."

defun proxy-norm-text (agent expr)
  "expr is a list representing a text typed in by the master. The function uses the local ontology to clean the text, removing empty words, replacing symbol by their canonical equivalents (taken from the ontology), and leaving data strings alone. It the result consists only of data strings, i.e., of all non-strings have been removed, then this is considered a failure and the expr is returned unprocessed.
Arguments:
agent: agent
expr: a list to be normalized."

defun proxy-parse-inform-message (agent text &aux input args to)
  "parse a text into a message structure. E.g.,"
albert watch the price increase of the technological bonds
(:date *now* :type :inform :from <agent> :to albert
 :action ()
 :args '(
 "watch the price increase of the technological bonds")

Arguments:
 agent: agent
text: text to be parsed.

defun proxy-parse-master-text (agent text)
 "parse the input text into some sort of structured list, taking into account the ontology, the master's model, the conversation context, or any other usable info. Currently acts as a noop.
Arguments:
 agent: agent
text: text as a list extracted from the master's area.

defun proxy-parse-request-message (agent text &aux input)
 "parse a text into a message structure. E.g.,
 ASK mul-1 MULTIPLY (2 3)
 (:date *now* :type :request :from <agent> :to mul-1
 :action 'MULTIPLY
 :args '(2 3))
Arguments:
 agent: agent
text: text to be parsed.

defun proxy-process-answer (agent item)
 "process an answer message found in the waiting-answer box of the assistant agent. We should process data, then kill pending-requests entry, then remove the answer message from the waiting-answers list.
Arguments:
 agent: agent
 item: answer message to be processed.

defun proxy-process-master-text (agent text &aux message)
 "processes the input found in the master input pane, removing characters that could pose problems for building a lisp list. Currently, take only parents into account.
Arguments:
 agent: agent
text: text string as extracted from the master input area.

defun proxy-save-answer (agent &rest ll)
 "saves a specific answer. Should start some sort of dialog to see what to do.

defun proxy-process-text-for-answer (agent text &aux message)
 "agent is waiting for an answer to a specific task. Parse text and, if it makes sense put the answer onto the saved-answer slot.

defun proxy-process-text-for-inform (agent text &aux message)
 "parse text and if tis makes sense sends the inform message."
defun proxy-process-text-for-request (agent text &aux
message)
  "get a string from master and builds a request message to
be inserted into the agenda of the assistant agent with the
action SEND-REQUEST. This will set-up a task to some other
agent or a broadcast task. The answer will be sent back to
the master.
Arguments:
  agent: agent
  text: text typed in by the user"

defun proxy-process-waiting-message (agent message)
  "function that picks up a waiting message in the waiting
messages bin and goes to process it, i.e., should produce
an answer and return it to sender.
Arguments:
  agent: agent
  message: request message to be processed by master."

defun proxy-reset-window (agent)
  "reset win-com windows for proxy agents. Called by reset.
Arguments:
  agent: agent."

defun proxy-revive-discarded-message (&rest ll)
  "should revive a discarded message."