UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique

MOSS 7 - Dialogs

Jean-Paul Barthès

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

Email: barthes@utc.fr

N225
July 2008

Warning

This document is describes the implementation of the dialog mechanism developed in the MOSS 6 version. The implementation also applies to the OMAS multi-agent platform and introduces new constraints in the type of dialogs that can be implemented.

A more complete version of the dialog process in particular in connection with a Personal Assistant agent can be found in
`N217L-OMAS v7 Personal Assistant`

A current research version of MOSS 7 runs in a MacIntosh Common Lisp environment (MCL 5.1 or 5.2 for OSX) and in an Allegro Common Lisp environment (ACL 6.1 and 8.1 running under Windows XP).

Keywords

Object representation, object-oriented programming environment, human machine interface, natural language dialogs

Revisions

Version	Date	Author	Remarks
1.0	Jul 08	Barthès	Initial issue

MOSS documents

Related documents

- UTC/GI/DI/N196L - PDM4
- UTC/GI/DI/N206L - MOSS 7 : User Interface (Macintosh)
- UTC/GI/DI/N218L - MOSS 7 : User Interface (Windows)
- UTC/GI/DI/N219L - MOSS 7 : Primer
- UTC/GI/DI/N220L - MOSS 7 : Syntax
- UTC/GI/DI/N221L - MOSS 7 : Advanced Programming
- UTC/GI/DI/N222L - MOSS 7 : Query System
- UTC/GI/DI/N223L - MOSS 7 : Kernel Methods
- UTC/GI/DI/N224L - MOSS 7 : Low Level Functions
- UTC/GI/DI/N225L - MOSS 7 : Dialogs
- UTC/GI/DI/N228L - MOSS 7 : Paths to Queries

Readers unfamiliar with MOSS should read first N196 and N219 (Primer), then N220 (Syntax), N218 (User Interface). N223 (Kernel) gives a list of available methods of general use when programming. N222 (Query) presents the query system and gives its syntax. For advanced programming N224 (Low level Functions) describes some useful MOSS functions. N209 (Dialogs) describes the natural language dialog mechanism.

Contents

1 Introduction

MOSS provides a general mechanism for developing natural language dialogs in the favorite language of the user. A small on line documentation application has been developed to illustrate the possibilities of the dialog mechanism. The MOSS user is interfaced through the MOSS window when in HELP mode (1).

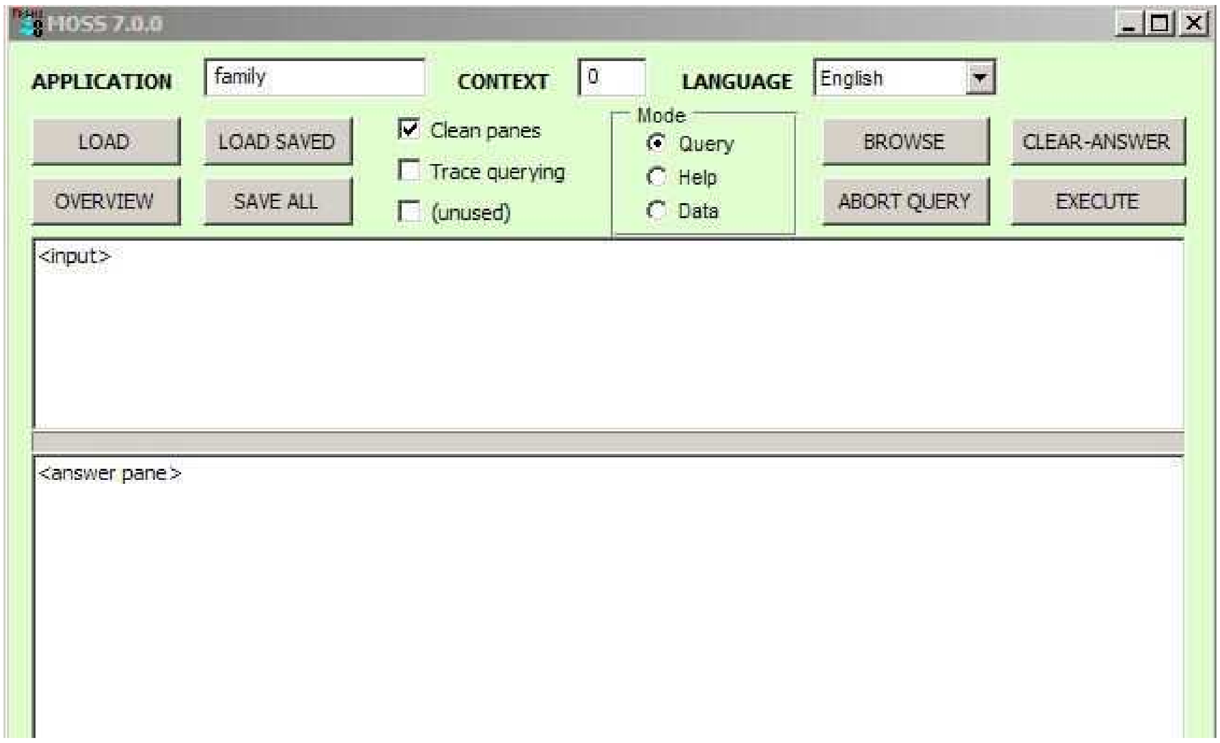


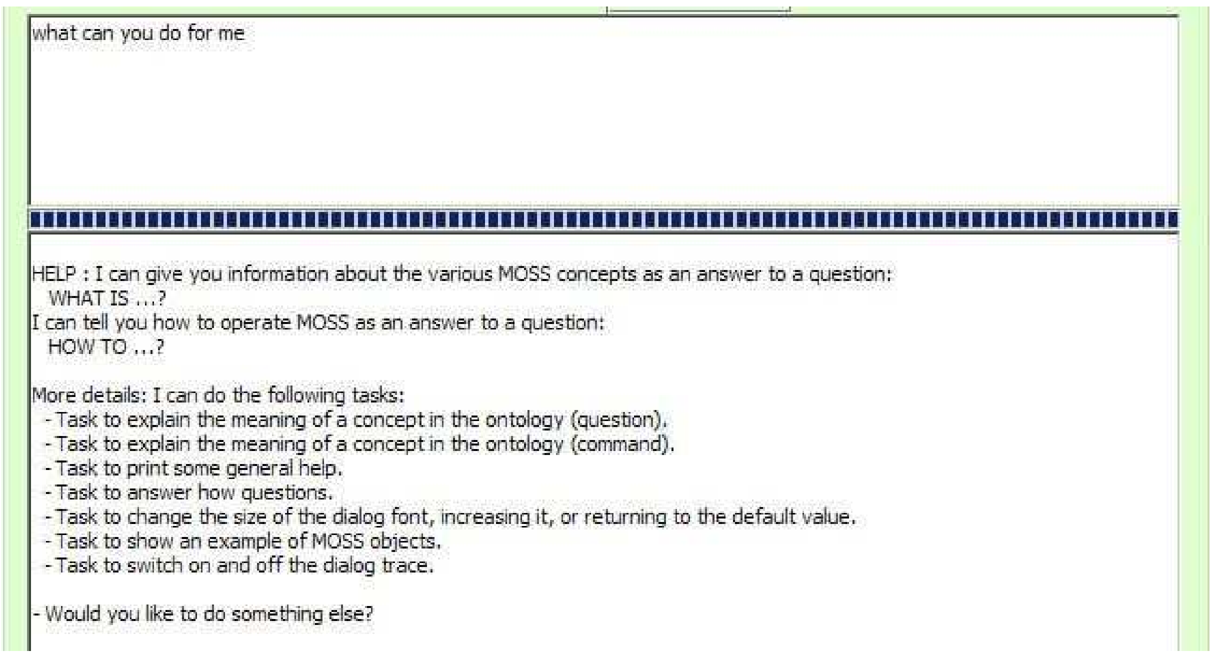
Figure 1: MOSS interaction window

Who should Read the Manual

This manual is intended for those who want to learn how a natural language interface was designed and built in the context of the MOSS environment. A more detailed manual is available for designing OMAS Personal Assistant dialogs.

2 Example of Dialog

The interaction between the user and MOSS occurs normally through the MOSS window (2) using the Hel mode. Under certain conditions it can be carried out in the Lisp listener, which is easier for obtaining a transcript.



```
what can you do for me

HELP : I can give you information about the various MOSS concepts as an answer to a question:
  WHAT IS ...?
I can tell you how to operate MOSS as an answer to a question:
  HOW TO ...?

More details: I can do the following tasks:
 - Task to explain the meaning of a concept in the ontology (question).
 - Task to explain the meaning of a concept in the ontology (command).
 - Task to print some general help.
 - Task to answer how questions.
 - Task to change the size of the dialog font, increasing it, or returning to the default value.
 - Task to show an example of MOSS objects.
 - Task to switch on and off the dialog trace.

- Would you like to do something else?
```

Figure 2: MOSS dialog in the interaction window (Help mode)

The following example is a transcript of a session. The dialog is however biased in the sense that it uses the documented parts of the MOSS system known to exist.

In the following dialog the user input is printed in italics.

```
? (moss::start-conversation moss::_main-conversation)
Warning. This is not a real dialog in the sense that every question usually does not
use the previous context. Thus, referents like pronouns or ellipsis (referring to part
of the answer) won't work.
- Hi! I will try to do my best to answer your question, but remember that my IQ is limited?
 What do you want to know?
- What is MOSS?

----------
     MOSS is an experimental system for modeling concepts. It can be used in particular
 to develop ontologies. MOSS has been first implemented in 1986, using a model
 called PDM (for Property-Driven Model). PDM has been refined and improved until
 today. PDM and MOSS were developed at UTC by Jean-Paul Barthès and PhD students.
----------
- What else can I do for you?
- what is a concept?

----------
     A CONCEPT is something representing an object of the real world. It can be
 abstract (an idea) or concrete (a person). If abstract, also said to be defined
 in intension, it is usually represented by a class (a MOSS ENTITY). When
 concrete, i.e. representing an individual, id is usually represented by an instance
 of some class. In MOSS, it could
 be represented as an ORPHAN, i.e. a classless object.
----------
- What else can I do for you?
- what is an orphan?

- I do not understand what you want to know. I can only give information about MOSS
 objects or MOSS procedures (HOW TO...).
- About what MOSS object do you want information? (if you want to quit, simply say: none).
- orphan?

- I do not understand what you want to know. I can only give information about MOSS
 objects or MOSS procedures (HOW TO...).
- About what MOSS object do you want information? (if you want to quit, simply say: none).
- none.

- Is there anything more I can do for you?
- help?

----------
     I can give you information about the various MOSS concepts as an answer to a question:
          WHAT IS ...?
      I can tell you how to operate MOSS as an answer to a question:
          HOW TO ...?
      If you do not want anything, I'll go to sleep...
----------
- Is there anything more I can do for you?
- what time is it?
- Current time is 9:3:11 and today is the 22/6/2005
- Would you like to know something else?
- yes.

- I'm listening....
- how do I create a class?

----------
     to create a CLASS (or MODEL) you can use the m-defclass macro or the m-make-class
 function.
      E.g.,
       (m-defclass PERSON
            (:att NAME (:entry)(:max 3))
            (:att AGE (:max 1))
            (:rel BROTHER PERSON))
      or
       (m-make-class 'PERSON
            '(:att NAME (:entry)(:max 3))
            '(:att AGE (:max 1))
            '(:rel BROTHER PERSON))
      The datailed syntax is the following:
        (m-defclass name &rest option-list) - creates a new class
      syntax of option list is
    (:is-a <class-id>*)         for defining inheritance
    (:tp <tp-description>) specifies terminal prop
    (:sp <sp-description>) specifies structural prop
    (:doc <documentation list>)
      Options for terminal properties are;:
    (:is-a <class-id> (:system <system-name>) )*
    for defining inheritance one class at a time
    (:min <number>) minimal cardinality
    (:max <number>) maximal cardinality
    (:name <var-name>) external variable-name
    (:default <value>) default, not obeying PDM2 specifications
    (:unique) minimal and maximal cardinality are each 1
    (:entry <function-descriptor>) specify entry-point
    if no arg uses make-entry, otherwise uses
    specified function
      where
    <function-descriptor>::=<arg-list><doc><body>
    e.g.  (:entry (value-list)(Entry for Company name)
      (intern(make-name value-list)))
      Options for structural properties are:
    (:is-a <class-id>*) for defining inheritance
    (:min <number>) minimal cardinality
    (:max <number>) maximal cardinality
    (:unique) minimal and maximal cardinality are each 1
    (:inv <prop-name>) inverse-property name

      Examples
    (:tp NAME (:min 1)(:max 2))
    (:sp AUTHOR BOOK (:inv WRITTEN-BY))
----------
- Is there anything more I can do for you?
```
- *how to create a method?*

```
----------
     MOSS has three kinds of methods:
      - own-methods
      - instance-methods
      - universal-methods
   Each type of methods has a corresponding creation macro and function.
----------
- Is there anything more I can do for you?
```
- *how to create an instance method?*

```
----------
     an INSTANCE-METHOD is a method that applies to instance of a given class
 as in any object-oriented environment. Creating a METHOD is done with the
 m-defmethod macro or the m-make-method function.
        E.g.,
         (m-defmethod =summary PERSON ()
             (HAS-NAME))
      Some remarks:
       - =my-print is the name of the method (MOSS convention is to start a method
 name with an = sign
       - PERSON is the name of the class
       - (HAS-NAME) is an accessor (built automatically by the system that allows
 to get the content of the HAS-NAME slot of the object.
----------
- Would you like information about something else?
```
- *give me an example of class.*

```
----------
     A class represents a concept, e.g., a PERSON, a BOOK, a PROPERTY. It gives
 the structure of objects that will be considered as its instances. However, an instance
 need not have the exact structure described in the class as in most of the OOL.
 Instance methods, attached to the class factor part of the behavior of an instance.
      Example:
        Let us define the class PERSON as follows:
           (m-defclass PERSON
                        (:att name (:entry) (:max 3))
                        (:att first-name)
                        (:rel brother person))
        stating that a PERSON has a NAME (index) a FIRST-NAME, may have a BROTHER.
        The system returns an internal identifier for the class, here $E-PERSON. Remember
 that the class is a MOSS object.
        Sending a message to the object yields:
          ? (send _person '=print-object)
          ----- $E-PERSON
          HAS-TYPE: $ENT
          HAS-ENTITY-NAME: PERSON
          HAS-RADIX: $E-PERSON
          IS-ENTITY-LIST-OF: MOSS
          HAS-COUNTER: 1
          HAS-TERMINAL-PROPERTY: HAS-NAME / PERSON, HAS-FIRST-NAME / PERSON
          HAS-STRUCTURAL-PROPERTY: HAS-BROTHER
          IS-SUCCESSOR-OF: HAS-BROTHER
          -----
          "*done*"
        which describes the class structure.
----------
- Is there anything more I can do for you?
```
- *no.*

```
- OK. I wait until you are ready.
- Wake me up when you want by typing something...
```
- *hi.*

```
Sorry I did not get that. Could you rephrase it? Or type: help.
```
- *how do I modify an object?*

```
----------
     it is possible to interactively modify the content of any object by locating it
 first and then using the EDITOR.
      Alternatively, a number of methods can be called on the object to add a value,
 a link, to remove a value, a property, etc. See Kernel Methods.
----------
- Would you like information about something else?
```
- *how do I start the editor?*

```
Warning. This is not a real dialog in the sense that every question usually does
 not use the previous context. Thus, referents like pronouns or ellipsis (referring
 to part of the answer) won’t work.
- Hi! I will try to do my best to answer your question, but remember that my IQ is
 limited? What do you want to know?
- ...
```

The next sections describe how the dialog is implemented.

3 Dialog Graphs

Dialogs are modeled by conversation graphs. They are finite state machines, equivalently oriented graphs. A dialog is represented by a path (succession of states). A dialog starts at an initial state with some data, and, as the dialog proceeds (i.e., more states are traversed), the context gets enriched with new data. A dialog ends either with a success or a failure. When some unusual and fatal condition occurs during the dialog, a throw to an `:error` label is generated by MOSS.

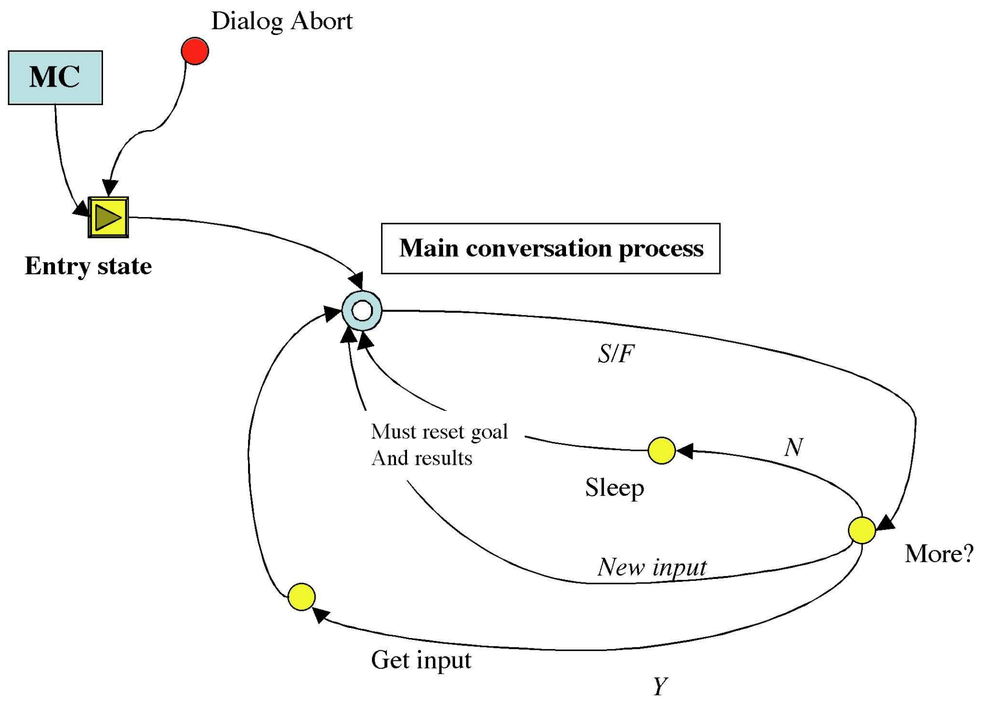


Figure 3: Example of conversation graph (OMAS Personal Assistant)

A state represents a conversation step. It usually corresponds to a pair <question, answer> or to a simple statement.

A specific graph can represent a partial conversation or "sub-dialog" that can be "called" from any other conversation. Each dialog or sub-dialog is indexed by a header.

The set of states represents all the possible dialogs on a particular topic. The set is fixed. Only the way the sets can be traversed varies from dialog to dialog

3.1 Dialog Graphs

A set of states is not physically implemented as a graph since there are no explicit arcs between two states. Transitions are computed dynamically. Thus, a set of states represents an ensemble of virtual dialog graphs.

3.2 State and Transitions

A state is implemented as an object, instance of a class called Q-STATE. Each state has an important property, namely its Q-CONTEXT. Q-CONTEXT is an object whose role is to gather information as the conversation is developed. Thus, when a transition occurs the context object associated to the new state is modified.

Most dialogs are goal oriented. In that case contexts can be seen as patterns to be progressively filled with information in order to gather enough data to select an action for reaching this goal.

3.3 Methods attached to a State

Since MOSS is an object environment, methods can be attached to a state. At least two methods must be present: (i) =execute ; and (ii) =resume. Any number of other methods can be attached to a particular state or context. In addition state objects use other methods like =make-transition.

=execute is executed when entering a state after a transition has occurred. It usually produces a statement or prints a question to the user, and waits for the answer.

=resume is executed when the user provides some information. It then analyses the text content, updates the context object and computes a transition.

4 Designing a Dialog

Designing a dialog is a non trivial operation. Indeed, one has first to write down all possible questions that a user can ask in a particular situation and then build the corresponding conversation graph. This last step uses a special dialog language.

The dialog language consists of a set of macros and functions for building the conversation graph. Macros and functions contain keywords that will indicate how to analyze the user's inputs, and how to update the context object. The language is defined so that one does not have to write the =execute and textsf=resume methods manually. Thus, a macro, defstate, produces the various states and associated =execute and =resume methods.

Once the dialogs are built, they are tested with naïve users. Their behavior is recorded and the dialogs are modified to better take into account their approach. The process is iterated until the designer is satisfied, i.e. until the dialog covers enough cases so that a naïve user can use it intuitively.

4.1 Example

We take examples from the dialogs implementing the online MOSS documentation.

4.1.1 Recording Possible Questions

The first step consists of listing some typical questions that the user will have in the context of the dialog. Since we are building an online help facility, one can expect that questions will resemble the following ones:

- What is MOSS?
- What is an entity?
- What is an orphan?
- How can I create a method?
- How can I modify an object?
- Etc.

4.1.2 Building an Initial Graph

From the set of questions, now try to imagine a possible conversation graph (Fig.4).

Fig.4 simply indicates that we enter with an entry state. At this state we send a greeting message to the user, record the answer and move to the state "Process input." At this point, we might detect something to do, in which case we transfer to an "Action" sub-dialog, or we can go to sleep if the user does not want to interact, or we can ask if there is more work to do, eventually transferring to the "Get input" state. If the analysis fails to trigger one of the previous transitions, then MOSS activates an ELIZA type dialog (eventually to catch marginal remarks, rather than stating "I did not understand").

4.1.3 Implementing the Conversation Graph

Once we have designed the graph we must build the corresponding structures: a dialog, the set of state, the transition rules.

- Dialog
  A dialog is built by providing a dialog header.

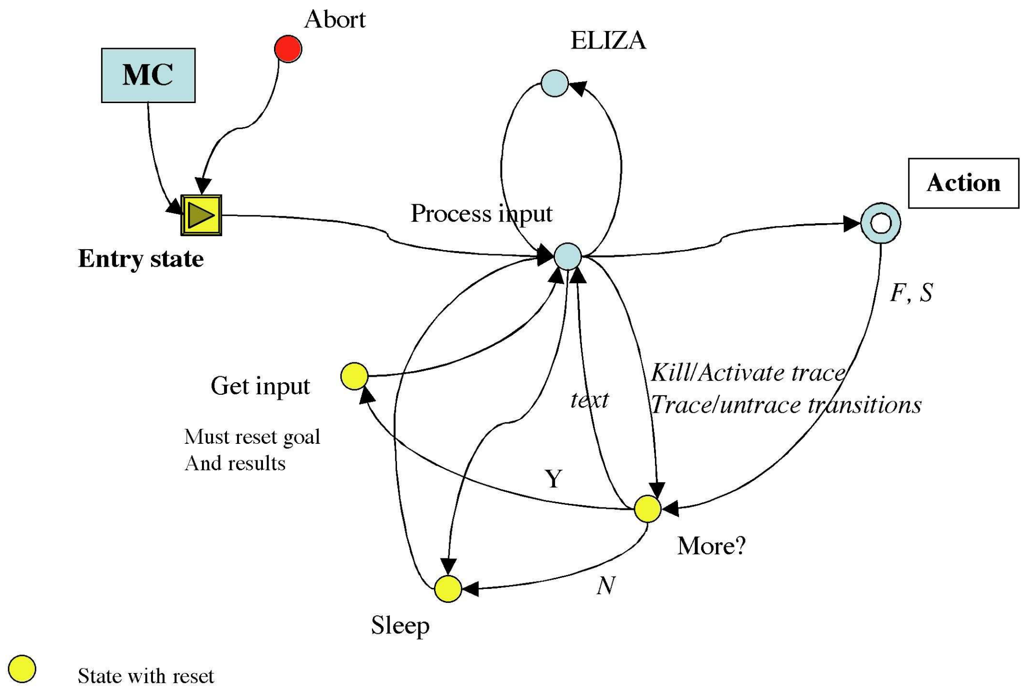


Figure 4: MOSS Main conversation graph

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

- Dialog States
  Dialog states are declared as global variables. This could be done automatically, but it is a way to view all the states, with some comments.

```
(defVar _q-abort () "Aborting state")
(defVar _q-eliza () "for doing small talk")
(defVar _q-entry-state () "entry to the state graph")
(defVar _q-get-input () "waiting for user input")
(defVar _q-more? () "asking for more work")
(defVar _q-process-input () "Processing the user input.")
(defVar _q-sleep () "Nothing to do, wait for user input")
```

  Note that a special ABORT state is defined to handle errors while processing data.
  Then each state must be implemented. To do this we use the `defstate` macro.

```
(m-defstate _q-entry-state
  (:label "Dialog start")
  (:explanation
   "Initial state when the assistant starts a conversation. ~
    Send a welcome message and wait for data.")
  (:reset-conversation)
  (:answer "Warning. This is not a real dialog in the sense that every question ~
           usually does not use the previous context. Thus, referents like pronouns ~
           or ellipsis (referring to part of the answer) won't work.")
  (:question '("~%- Hello! What can I do for you?"
               "~%- Hi! What would you like to know?"
               "~%- Hi! I will try to do my best to answer your question, but ~
            remember that my IQ is limited? What do you want to know?"))
  (:prompt "-")
  (:transitions
   (:always :save :target _q-process-input))
  )
```

Let us review some or the options of the `defstate` macro.

A state may have a label indicating which state it is. Here: ”Dialog start.” An explanation may be provided concerning the state role. The `:reset-conversation` option does some cleaning, in particular clearing the context. The `:answer` option simply posts the corresponding text. The `:question` option prints the corresponding text. The :prompt option sets the prompting characters. The `:transition` option is more complex: here it simply saves the user answer and makes a transition to the ”Process input” state.

Note that no analysis of the user input is done in this state. The analysis occurs in the ”Process input” state.

```
(m-defstate _q-process-input
  (:label "Process initial data")
  (:explanation
   "Process what the user said that is stored into the HAS-DATA slot of Q-CONTEXT.")
  (:no-execute)
  (:input-from-data)
  (:pattern-var ?x)
  (:transitions
   (:patterns (("what" "time" (?* ?x)))
              :exec (print-time) :target _q-more?)
   (:patterns (("what" "is" (?* ?y))
               ("what" "are" (?* ?y))
               ((?* ?x) "explain" (?* ?y))
               ((?* ?x) "meaning" (?* ?y))
               ((?* ?x) "mean" (?* ?y))
               ((?* ?x) "means" (?* ?y)))
              :keep ?y
              :sub-dialog _what-conversation
              :success _q-more? :failure _q-more?)
   (:patterns (("help")
               ((?* ?x) "use" "MOSS" (?* ?y))
               ("what" "questions" (?* ?x))
               ((?* ?x) "tell" "me"))
              :replace '("help")
              :sub-dialog _what-conversation
              :success _q-more? :failure _q-more?)
   (:patterns (((?* ?x) "show" "me" (?* ?y) "example" (?* ?z))
               ((?* ?x) "give" "me" (?* ?y) "example" (?* ?z)))
              :keep ?z :sub-dialog _show-example-conversation
              :success _q-more? :failure _q-more?)
   (:patterns (((?* ?x) "show" (?* ?y))
               ((?* ?x) "list" (?* ?y))
               ((?* ?x) "print" (?* ?y))
               ((?* ?x) "display" (?* ?y)))
```

```
                :keep ?y
                :sub-dialog _print-conversation
                :success _q-more? :failure _q-more?)
   (:patterns (((?* ?x) "no" "trace" (?* ?y))
               ((?* ?x) "kill" "trace" (?* ?y))
               ((?* ?x) "trace" "off" (?* ?y))
               ((?* ?x) "stop" "trace" (?* ?y))
               ((?* ?x) "no" "verbose" (?* ?y)))
              :exec (setq *verbose* nil) :target _q-more?)
   (:patterns (((?* ?x) "trace" "on" (?* ?y))
               ((?* ?x) "set" "trace" (?* ?y))
               ((?* ?x) "start" "trace" (?* ?y))
               ((?* ?x) "verbose" (?* ?y)))
              :exec (setq *verbose* t) :target _q-more?)
   (:patterns (((?* ?x) "show" "transitions" (?* ?y))
               ((?* ?x) "trace" "transitions" (?* ?y))
               ((?* ?x) "transitions" "on" (?* ?y))
               ((?* ?x) "print" "transitions" (?* ?y)))
              :exec (setq *transition-verbose* t) :target _q-more?)
   (:patterns (((?* ?x) "hide" "transitions" (?* ?y))
               ((?* ?x) "transitions" "off" (?* ?y))
               ((?* ?x) "do" "not" "show" "transitions" (?* ?y)))
              :exec (setq *transition-verbose* nil) :target _q-more?)
   (:starts-with ("how") :trim-data (input) :sub-dialog _how-conversation
                 :success _q-more? :failure _q-more?)
   (:contains ("nothing") :trim-data (input) :target _q-sleep)
   (:otherwise :target _q-ELIZA))
  )
```

The previous expression might look deceivingly complex. It is not really so. The `:no-execute` option means that no question being asked no =execute method needs to be generated. The `:input-from-data` option indicates that the data to be analyzed comes from the data part of the context. The `:pattern-var` indicates that ?x is a variable occurring in a pattern whose content might be used after the pattern has been applied. The `:transition` option includes a number of options: the patterns options are detailed thereafter; the `:starts-with` option indicates what to do if the text starts with the word ”how”; the `:contains` option indicates what to do if the text contains ”nothing” (the two cases could be processed with patterns).

- Patterns
  The `:patterns` option has a somewhat complex syntax. Its role is to specify a list of patterns to be applied to the data being processed. Patterns are applied in the order. When a pattern applies, then the rest of the patterns option is executed. A pattern is modeled on the ELIZA approach as reported by Norvig ([Nor92]). No stemming is done on the input, nor any ontological filtering. The options used above are the following:

  :execute simply executes the Lisp expression (provides a hook to internal options)

  :keep ¡variable¿ means that we keep the content of the variable, usually part of the sentence

  :target ¡state¿ means that if successful we make a transition to the corresponding state

  :subdialog ¡conversation¿ means that we start a subdialog. Returning from a subdialog needs two addresses given respectively by :success ¡state¿ and :failure ¡state¿.

It can be seen from the list of patterns that any text from the user starting with ”what is” or ”what are” or containing ”explain” ”meaning” ”mean” ”means” will trigger a ”What conversation,” i.e. a subdialog for processing the question. Any data containing ”show” ”list” ”print” or ”display” will trigger a ”Print conversation.” Etc.

We give the other states for information.

```
(m-defstate _q-abort
  (:label "Aborting")
  (:explanation "Something went wrong, ABORT the dialog and restart it.")
  (:no-immediate-abort-test) ; otherwise we enter a loop
  (:immediate-transitions
   (:always :reset))
  (:no-resume))

(moss::m-defstate _q-ELIZA
  (:label "ELIZA")
  (:explanation
   "Whenever MOSS cannot interpret what the user is saying, ELIZA is called to do
   some meaningless conversation to keep the user happy.")
  (:eliza)
(:transitions
   (:always :save :target _q-process-input)))

(m-defstate _q-GET-INPUT
  (:label "Get input")
  (:explanation
   "State in which the user inputs his request.")
  (:reset)
  (:question "~%- I'm listening....")
  (:transitions
   (:always :save :target _q-process-input)))

(m-defstate _q-more?
   (:label "More?")
   (:explanation "Asking the user it he wants to do more interaction.")
   (:reset-conversation)
   (:question
    '("~%- What else can I do for you?"
      "~%- Is there anything more I can do for you?"
      "~%- Would you like to know something else?"
      "~%- Would you like information about something else?")
    )
   (:transitions
    (:starts-with (nothing) :target _q-sleep)
    (:no :target _q-sleep)
    (:yes :target _q-get-input)
    (:otherwise :save :target _q-process-input)))

(m-defstate _q-SLEEP
  (:label "Nothing to do")
  (:explanation
   "User said she wanted nothing more.")
  (:reset)
  (:answer "~%- OK. I wait until you are ready.")
  (:question "~%- Wake me up when you want by typing something...")
  (:transitions
   (:always :save :target _q-process-input))
  )
```

When multiple lines occur for example in a :question option, this means that one of the lines is chosen randomly, in such a way to vary answers to the user.

- Conversations as Subdialogs
  Subdialogs are constructed in the same fashion. Let us examine the case of the "What conversation."

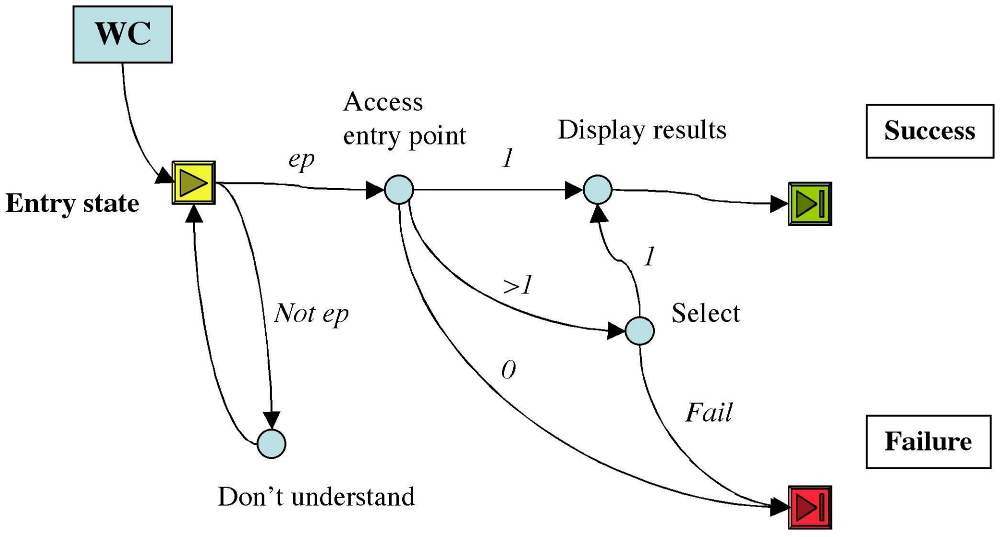


Figure 5: What is conversation (subdialog)

As can be seen Fig.5 a subdialog graph has an entry state, a success exit state and a failure exit state. In addition an abort mechanism allows a throwing action.

The difference with the main dialog is that this dialog is oriented towards fulfilling an action, namely displaying results associated with the "What is" question. In order to trigger the action, we associate an action pattern with the subdialog, while defining the dialog header. This is done as follows:

First we define a specific type of action, a WHAT action:

```
(m-defclass
 WHAT-ACTION
 (:is-a ACTION)
 (:id $WAPE)
 (:att OPERATOR (:default display-documentation))
 (:rel OPERATOR-ARGUMENTS MOSS-OBJECT-ARGUMENT)
 (:doc "WHAT-ACTION is a pattern that has a single argument that must be a MOSS object.
 When complete the system will display the content of the HAS-DOCUMENTATION property
 of the MOSS object.")
 (:var _what-action))
```

Then, we define the corresponding piece of dialog as follows:

```
(m-defdialogmixin
 WHAT
 (:short-name wc)
 (:label "What conversation")
 (:explanation
  "This dialog is meant to give some information to the user regarding the various
  entities of the MOSS system. The idea is to get the identifier of an object and to prin
  the HAS-DOCUMENTATION slot associated with it. If not present we call the =what?
   method.)
 (:action _what-action) ; points towards the action subclass
 (:failure-explanation "could not understand what object the user wanted to create.")
 (:success-explanation "success exit with an oid in the results slot"))
```

the m-defdialogmixin macro allows to define a dialog that can be called as a subdialog. I.e., it will produce the success and failure output states.

The states of this subdialog illustrate additional mechanisms of the dialog language.

Entry state:

```
(m-defstate
 _wc-entry-state
 (:label "start of the WHAT dialog")
 (:explanation
  "Initial state may have data left in the q-context. The data either represent entry poi
  something else that should be used to build a query.")
 (:exec-var test-result)
 (:immediate-transitions
  (:patterns (((?* ?x) "terminal" "property" (?* ?y)))
             :set-results 'TERMINAL-PROPERTY :target _wc-access-ep)
  (:patterns (((?* ?x) "structural" "property" (?* ?y)))
             :set-results 'STRUCTURAL-PROPERTY :target _wc-access-ep)
  (:patterns (((?* ?x) "instance" "method" (?* ?y)))
             :set-results 'INSTANCE-METHOD :target _wc-access-ep)
  (:patterns (((?* ?x) "own" "method" (?* ?y)))
             :set-results 'OWN-METHOD :target _wc-access-ep)
  (:patterns (((?* ?x) "universal" "method" (?* ?y)))
             :set-results 'UNIVERSAL-METHOD :target _wc-access-ep)
  (:special-test :res-var test-result :method '=test :args (conversation)
                 :set-results (car test-result) ; take the first ep
                 :target _wc-access-ep)
  (:patterns (((?* ?x) "none"  (?* ?y))
              ((?* ?x) "forget"  (?* ?y))
              ((?* ?x) "give" "up"  (?* ?y)))
             :target _wc-failure)
  (:otherwise :target _wc-dont-understand))
 (:no-resume)
 )
```

The entry state does includes new features

The `:no-resume` option that indicates that no question being put to the user, no analysis of the answer is required.

The `:special-test` option allows to define a special test as a method that will be associated with the conversation. Here the test simply checks if there are any entry point in the data by combining several words if needed.

The "Access entry point" accesses the objects corresponding to the entry point. The corresponding state is:

```
(m-defstate
 _wc-access-ep
 (:label "Got an entry point")
 (:explanation
  "We have an entry point and can use it to access the knowledge base by calling ~
   the access function.")
 (:manual-execute)
 (:no-resume))
```

The :manual-execute option allows to write the =execute method manually. Here the method accesses the object base and puts the results into the HAS-RESULTS slot of the context object as follows:

```
(m-defownmethod
 =execute _wc-access-ep (conversation)
 "Using entry point to access objects. If one, OK, if more, must ask user to ~
  select one."
 (let* ((state-context (car (HAS-Q-CONTEXT conversation)))
        (entry-point (car (HAS-RESULTS state-context)))
        results)
   (verbose-format "~%entry point in data => ~S" entry-point)
   ;; access objects
   (setq results (access entry-point :verbose *verbose*))
   (verbose-format "~%results of access => ~S" results)
   ;; save results
   (setf (HAS-RESULTS state-context) results)
   ;; if one value, success, otherwise must ask the user to choose
   (if (cdr results)
     (send *self* '=set-transition conversation _wc-select)
     (send *self* '=set-transition conversation _wc-display-results))))
```

If more than one object is found, then the transition is done onto the "Select" state, otherwise we transfer control to the "Display" state.

The "Display" state has the following structure:

```
(m-defstate
 _wc-display-results
 (:label "WC display results")
 (:explanation "we got an object identifier and display the associated doc or we send a
 =what? method to the object.")
 (:manual-execute)
 (:question "~%So?..."))
```

Here, the =execute method is inserted manually:

```
(m-defownmethod
 =execute _wc-display-results (conversation)
 "save the obj-id of the result into the action pattern and executes it."
 (let* ((state-context (car (HAS-Q-CONTEXT conversation)))
        (pattern (car (HAS-GOAL conversation)))
        (obj-arg (car (HAS-operator-arguments pattern)))
        (obj-id (car (HAS-RE\begin{verbatim}
```

Here, the =execute method is inserted manually:

```
(m-defownmethod
 =execute _wc-display-results (conversation)
 "save the obj-id of the result into the action pattern and executes it."
 (let* ((state-context (car (HAS-Q-CONTEXT conversation)))
        (pattern (car (HAS-GOAL conversation)))
        (obj-arg (car (HAS-operator-arguments pattern)))
        (obj-id (car (HAS-RESULTS state-context)))
        )
   ;; add value to the arg
   (send obj-arg '=replace 'HAS-ARG-VALUE (list obj-id))
   ;;execute pattern
   (send pattern '=activate)
   ;; indicate success in results
   (setf (HAS-RESULTS state-context) '(:success))
   ;; transition to success
   (send *self* '=set-transition conversation _wc-success)
   ))
```

When the method is executed, the action pattern is filled and the action is launched by sending an =activate message to the pattern. Then, we exit with a transition onto the "Success" state.

The "Don't understand" state has the following structure:

```
(m-defstate
 _wc-dont-understand
 (:label "WC arguments not understood")
 (:explanation "we have some data but there is no entry point among them.")
 (:answer "~%- I do not understand what you want to know. I can only give information abou
 (:question "~%- About what MOSS object do you want information? (if you want to quit, si
 (:transitions
  (:always :save :target _wc-entry-state)))
```

Some message is printed and a transition occurs always to the entry-state.

The "Select" state is used when more than one object corresponds to the entry point

```
(m-defstate
 _wc-select
 (:label "wc select")
 (:explanation "we have retrieved more than one object we ask user to select one.")
 (:ask-TCW :title "Select one of the objects"
           :choice-list (mapcar #'(lambda (xx) (list (send xx '=summary) xx)) results)
           :why "Select the object that you want documentation about.")
 ;where do we put the result? in to-do, faking a question
 (:transitions
  (:otherwise :set-context ('HAS-RESULTS input)
              :target _wc-display-results)))
```

the :ask-TCW option posts a selection window displaying a list ob objects among which the user must select one.

The "Success" and "Failure" states are synthesized automatically.

Other subdialogs are developed in the same way. For example the "How" dialog has the following structure shown in Fig.6. It can be seen that subdialogs are called corresponding to the type of action to comment upon: create, query, kill, delete, modify.

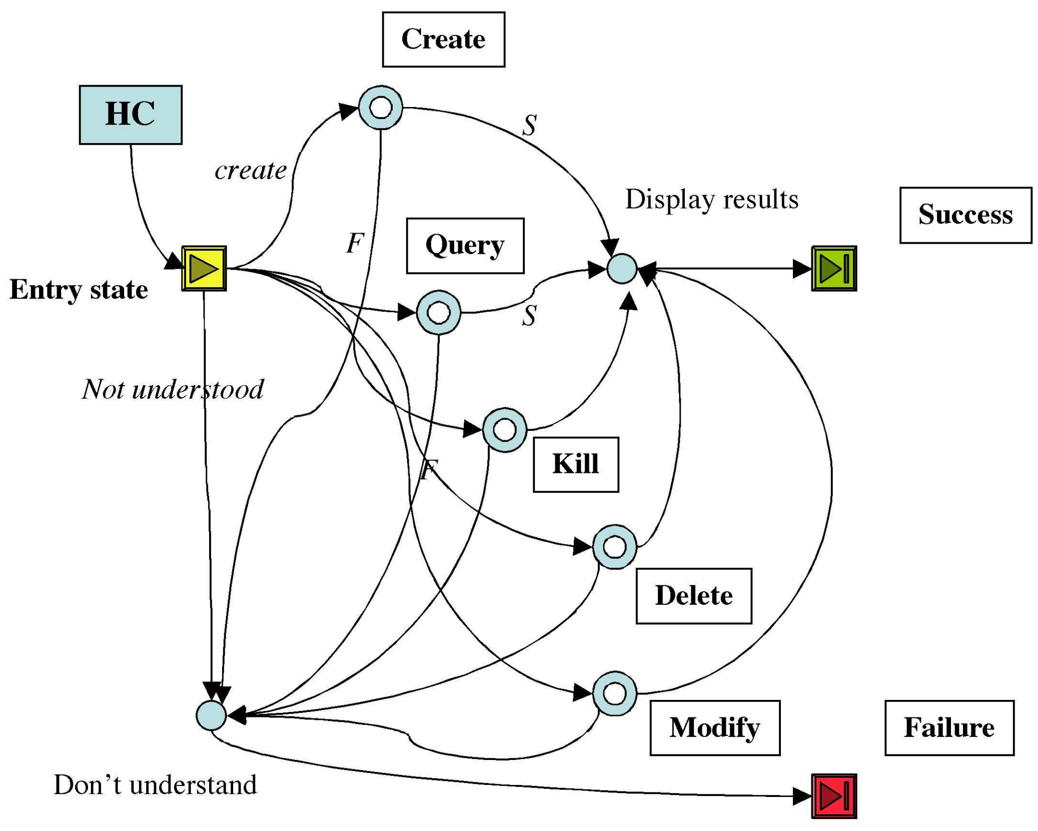


Figure 6: Graph of the "How" subdialog

The other subdialogs have analogous structure.

Most of the dialog graph construction relies on a few macros and functions that are described in the following sections.

5 Dialog Process

The dialog process consists of a number of functions that traverse the conversation graph.

5.1 Initialization

When a dialog is started by clicking the Help button of the MOSS window, a new thread is created (converse process) that activates the moss::start-conversation function specifying the I/O channels and the initial conversation header (beginning of the dialog).

The moss::start-conversation function creates the various structures that are used in the dialog, namely:

- a conversation object (Q-CONVERSATION)
- a state-context object (Q-STATE-CONTEXT)

The function initializes the created structures, recording the initial state, the current state, the conversation header and the newly created state context into the conversation object. The moss::start-conversation function then calls the dialog-engine function.

5.2 The Conversation Algorithm

The dialog-engine function checks the validity of the conversation header and calls the dialog-state-crawler function that will traverse the different conversation states. The dialog-engine function also catches all fatal errors detected during the conversation and restarts a new dialog when a problem occurs.

Control of a conversation turn is done by the dialog-state-crawler function that works as follows.

- If a state is given in input, then it starts from this state, otherwise it start from the initial state of the conversation.
- It then enters a loop, getting out of it only when a fatal error occurs, in which case the dialog is reset.

5.3 Dialog Loop

During the dialog loop we hop from state to state. When entering a state, the =execute method is called to analyze the content of the data attached to the state context. After that a transition to another state may occur, or the master may be asked to contribute some more input (clarification, additional information). If we stay on the same state, then the =resume method is called to analyze the input from the master (or eventually some info coming from other agents). The details are given in the next paragraphs.

At the beginning of the loop we have a state. If the state is not a state object but one of the keywords `:failure` or `:success`, then we are in the process of returning from a sub-conversation and we do a special transfer by calling the return-from-sub-dialog function (see).

Otherwise, we record the state and make a transition to state using the make-transition function that returns the id (identifier) of the new state.

We then send an =execute message to the new state, which will trigger the processing attached to this state and return one of the following lists:

```
(:failure) (:success) (:transition <state>) (:resume)
(:resume-no-wait) (:sub-dialog <sub-dialog-info>)
```

The meanings are:

- failure or success mean that we are returning from a sub-dialog
- transition indicates an immediate transition to the specified state

- resume means that we must wait for a master?s input or answer from other agents before executing the resume method
- resume-no-wait means that we must execute the =resume method without waiting for a master?s input
- sub-dialog means that we want to enter a sub-dialog

On failure, success, transition we go back to the beginning of the loop.

On sub-dialog, we apply a special transition calling the make-transition-to-sub-dialog function.

On a resume action we wait for an input from the master or from other agents we may have called. The *process waits until something appears in the TO-DO slot of the conversation object.* After that we send a =resume message to the state.

On any other value we restart the dialog.

When the =resume message is sent to the state the returned values are:

```
(:failure) (:success) (:transition <state>) (:sub-dialog <sub-dialog-info>)
```

The meanings are:

- failure or success mean that we are returning from a sub-dialog
- transition indicates a transition to the specified state
- sub-dialog means that we want to use a sub-dialog

On failure, success, transition we go back to the beginning of the loop.

On sub-dialog, we apply a special transition calling the make-transition-to-sub-dialog function.

On any other value we restart the dialog.

5.4 Sub-dialogs

During the execution of a dialog, a sub-dialog can be called. In that case we call the make-transition-to-sub-dialog function. The function installs a frame entry in the frame-list stack of the conversation object. The frame entry has the form:

```
(success-state failure-state state goal)
```

where

- success-state is the state to transfer to on a success return, it may be one of the keywords `:success` or `:failure`
- failure-state is the state to transfer to on a failure return, it may be one of the keywords `:success` or `:failure`
- state is the current state
- goal is the current conversation goal if any, that will be reinstalled upon return.

A transition occurs onto the entry-state of the sub-dialog.

A sub-dialog usually has a goal represented by an action object that will be activated in case of success.

When returning from a sub-dialog the conversation object is restored. The frame-list is popped. If the sub-dialog is successful it will make a transition to the state specified by the success keyword of the dialog-state-crawler, if it fails it will make a transition to the state specified by the failure keyword. Examples are given in Section 5.

5.5 Dialog Goals and Actions

A sub-dialog usually includes a goal containing an action to be activated. The action is modeled by an action object containing a function name and descriptions of arguments. One may think of a Web Service where the function name would be the name of the service and the arguments the description of the input arguments. The description of the arguments can contain information like a question to be asked to the master in case the argument is compulsory and there was not enough information to fill it.

Running the sub-dialog is meant to fill at least the compulsory arguments of the action, and when this is done, the action is activated, closing the conversation segment.

6 Dialog Objects

The dialog structure is made of several types of objects: conversation-headers, state objects, context objects. The structure is built by using some macros like defstate or defdialogmixin. Objects are described in this section macros are described in the next section.

6.1 Conversation Header Objects

A conversation header is an object used to identify a conversation, i.e., a set of states one of which being the entry state to the conversation. It is a simple object:

```
----- $QHDE
 CONCEPT-NAME: Q-DIALOG-HEADER
 RADIX: $QHDE
 DOCUMENTATION: An object used to identify a dialog.
 ATTRIBUTE : LABEL/CONCEPT, EXPLANATION/Q-STATE
 RELATION : ENTRY-STATE/Q-DIALOG-HEADER, PROCESS-STATE/Q-DIALOG-HEADER,
     Q-ACTION/Q-DIALOG-HEADER
 COUNTER: 27
```

It can be seen that a conversation header object has a label, an explanation, and points to an entry-state and possibly a model of action.

Examples of conversation headers are:

```
----- $QHDE.1
 LABEL: Main conversation
 EXPLANATION: This is the main conversation loop.
 ENTRY-STATE: Dialog start
 PROCESS-STATE: Find performative
-----
```

or

```
----- $QHDE.5
 LABEL: Show example conversation
 EXPLANATION: This dialog is meant to give some examples of what one can do in the
 MOSS system.
 ENTRY-STATE: start of the SHOW EXAMPLE dialog
 ACTION: HOW-ACTION
-----
```

6.2 State Objects

State objects represent a node of the conversation graph.

```
----- $QSTE
 CONCEPT-NAME: Q-STATE
 RADIX: $QSTE
 DOCUMENTATION: A STATE models a particular state of a state graph which defines
  possible sequences corresponding to a plan and to a dialog. A state graph implements
  a complex skill of an agent.
 ATTRIBUTE : STATE-NAME/Q-STATE, LABEL/CONCEPT, EXPLANATION/Q-STATE
 INSTANCE-METHOD: =SUMMARY INST/ Q-STATE
 COUNTER: 30
-----
```

A state has a name, a label, an explanation, and two methods: =set-transition and =summary. There are two subclasses: failure states and success states.

Examples:

```
----- $QSTE.1
 LABEL: Dialog start
 EXPLANATION: Initial state when the assistant starts a conversation. Send a welcome
   message and wait for data. Also entered on a restart following an abort.
-----
```

or

```
----- $QSTE.4
 LABEL: Get input
 EXPLANATION: State in which the user inputs his request.
-----
```

6.3 Context Objects

Context objects model the information attached to a particular state of the conversation.

```
----- $SCXTE
 CONCEPT-NAME: Q-CONTEXT
 RADIX: $SCXTE
 DOCUMENTATION: defines the context corresponding to a state in a state graph. ~
             The context can contain any number of properties.
 ATTRIBUTE : DATA/Q-CONTEXT, performative/Q-CONTEXT, QUERY/Q-CONTEXT,
     WORKING-LIST/Q-CONTEXT, RESULTS/Q-CONTEXT, STATE-OBJECT/Q-CONTEXT,
     required task args/Q-CONTEXT, optional task args/Q-CONTEXT
 RELATION : NEXT-CONTEXT/Q-CONTEXT, TASK/Q-CONTEXT, TASK-LIST/Q-CONTEXT
 INSTANCE-METHOD: =SUMMARY INST/ Q-CONTEXT
 COUNTER: 96
-----
```

A *context object* contains the data used by a specific conversation process. It has several slots:

- DATA contains the data to be analyzed
- QUERY unused
- RESULTS contains results of the local analysis (used also for returning the results of a sub-dialog)
- STATE-OBJECT refers to the state to which the context applies
- LABEL is a simple label

State objects are linked by a NEXT-STATE property.

An example of state for processing the ”How do I create a class?” request:

```
----- $SCXTE.38
 performative: REQUEST
 STATE-OBJECT: $QSTE.46
 NEXT-CONTEXT:
     MOSS::$SCXTE.39 label: NIL
     DATA : (how do I create a class)
     RESULTS:
     WORKING-LIST: NIL
-----
```

6.4 Conversation Objects

A *conversation* object makes the link between the user system and the conversation system. It keeps track of a conversation.

```
----- $CVSE
 CONCEPT-NAME: Q-CONVERSATION
 RADIX: $CVSE
 DOCUMENTATION: An object whose instances represent a conversation and points to the ~
             stage of this conversation. A new conversation starts with the ~
             creation of an instance of this class, e.g. after a reset.
 ATTRIBUTE : AGENT/Q-CONVERSATION, OUTPUT-WINDOW/Q-CONVERSATION,
     INPUT-WINDOW/Q-CONVERSATION, TO-DO/Q-CONVERSATION,
     FRAME-LIST/Q-CONVERSATION, TEXT-LOG/Q-CONVERSATION
 RELATION : STATE/Q-CONVERSATION, INITIAL-STATE/Q-CONVERSATION,
     DIALOG-HEADER/Q-CONVERSATION, SUB-DIALOG-HEADER/Q-CONVERSATION,
     Q-CONTEXT/Q-CONVERSATION, GOAL/Q-CONVERSATION, task-list/Q-CONVERSATION
 INSTANCE-METHOD: =DISPLAY-TEXT INST/ Q-CONVERSATION,
     =RESET INST/ Q-CONVERSATION
 COUNTER: 2
-----
```

A conversation object contains references to the input/output windows, to data, to goals. It contains a stack of sub-conversation (frame-list) information about the current state and associated context, and about the current subdialog being executed.

Example:

```
----- $CVSE.1
 OUTPUT-WINDOW: #<OMAS-ASSISTANT-PANEL "ALBERT" #x52CACAE>
 INPUT-WINDOW: #<OMAS-ASSISTANT-PANEL "ALBERT" #x52CACAE>
 FRAME-LIST: ($QSTE.42 $QSTE.42 $SCXTE.3 NIL NIL)
 STATE: Début du dialogue
 INITIAL-STATE: Début du dialogue
 DIALOG-HEADER: $QHDE.13
 SUB-DIALOG-HEADER: $QHDE.14
 Q-CONTEXT:
     ALBERT::$SCXTE.967 label: NIL
     DATA : NIL
     RESULTS:
     WORKING-LIST: NIL
-----
```

6.5 Action and Argument Objects

The *ACTION class* only provides a minimal skeleton that must be sub-classed for defining actual actions.

```
----- $ACTE
 CONCEPT-NAME: ACTION
 RADIX: $ACTE
 DOCUMENTATION: Action represents all actions that can be executed by MOSS. For example, ~
             explaining a term or detailing a procedure. It contains an operator ~
             reference (function name) and a list of arguments to be given as ~
             keywords when calling the function.
 ATTRIBUTE : ACTION-NAME/ACTION, DOCUMENTATION/METHOD, OPERATOR/ACTION
 RELATION : OPERATOR-ARGUMENTS/ACTION
 OWN-METHOD: =CREATE-PATTERN OWN/ ACTION
 INSTANCE-METHOD: =SUMMARY INST/ ACTION, =ACTIVATE INST/ ACTION,
     =DISPLAY-DOCUMENTATION INST/ ACTION, =DISPLAY-OBJECT INST/ ACTION,
     =READY? INST/ ACTION
 COUNTER: 1
-----
```

A possible subclass is that defined for the HOW-ACTION:

```
----- $HAPE
 CONCEPT-NAME: HOW-ACTION
 RADIX: $HAPE
 DOCUMENTATION: HOW-ACTION is a pattern that has a single argument that must be
  a MOSS object. When complete the system will display the content of the
  HAS-DOCUMENTATION property of the corresponding documentation object.
 ATTRIBUTE: HAS-OPERATOR
 RELATION: HAS-OPERATOR-ARGUMENTS
 IS-A: ACTION
 OWN-METHOD: =CREATE-PATTERN
 INSTANCE-METHOD: =ACTIVATE, =READY?
 COUNTER: 7
-----
```

Arguments are also modeled for defining action patterns:

```
----- $ARGE
 CONCEPT-NAME: ARGUMENT
 RADIX: $ARGE
 DOCUMENTATION: ARGUMENT represents one of the arguments that must be filled without ~
            which an action or a task cannot be launched. The requirement however ~
            is not strict since the argument may be optional.
   An argument contains a name, one or more questions to ask the user when it is ~
            missing, a validation function, room for a value, it has a name.
 ATTRIBUTE : ARG-NAME/ARGUMENT, ARG-KEY/ARGUMENT, ARG-VALUE/ARGUMENT,
     ARG-QUESTION/ARGUMENT
 INSTANCE-METHOD: =SUMMARY INST/ ARGUMENT
 COUNTER: 1
-----
```

Arguments are organized as a hierarchy of types. See appendix on modeling actions for more details.

7 Dialog Language

The dialog language consists of a mixture of macros and functions or methods to be added by the user.

7.1 The `defstate` macro

The `m-defstate` macro is fairly complex and defined the dialog language. The use of the various options is non trivial.

When a state is entered, the =execute method is applied to the new state and associated context. If the state asks a question to the user (`:question`, `:ask`, `:ask-tcw`, `:ask-tsw` options), then processing the answer is done by calling the =resume method. When the state does not ask a question to the master, then the =resume method can be omitted, and a transition is specified in the =execute method.

7.1.1 The =execute method

The method normally processes the content of the HAS-DATA slot of the Q-CONTEXT object that was transferred from the previous state. We can also select the HAS-RESULTS slot. If the processing is successful a transition occurs onto some next state (i.e. the method returns the list (:transition <next-state>)). Otherwise, we may ask a question to the user and return the list (:resume) to wait for the answer and analyze it. Eventually, we may throw to the `:abort-dialog` tag if we are lost or found a serious error.

A number of options can be specified during execution of the =execute method.

- `:action` method args
  The specified method is applied to *self* (current state) with the specified args
- `:answer` string
  Displays a text to the master using the =display-text method of the conversation object. Answer displays a text but does not wait for a user input. The `:question` option can be used after that, however, this does not make much sense.
- `:answer-type` performative
  Puts the specified performatives into the state context object.
- `:ask` string
  Ask master, does not erase the screen.
- `:eliza`
  Switches to an ELIZA dialog, using the rules found in the dialog file.
- `:exec` expression
  The specified expression is executed.
- `:init-data {word}+`
  Used to put some initial data into the HAS-DATA slot of the Q-CONTEXT. Sets the internal DATA variable.
- `:manual-execute`
  The =execute method is hand coded for this state.
- `:question` text
  The associated value is either a string or a list of strings. When we have a list of strings one is chosen randomly for printing. Once the question is asked, the user is supposed to answer. The answer is inserted into the TO-DO slot of the conversation object that is the link between the external system and MOSS. The question is processed by a =resume method.
- `:question-no-erase` text
  Same as `:question` but does not erase the screen before displaying the question text.

- `:immediate-execute` function args
  the specified function is applied to the specified args.

- `:exec` expression
  the specified expression is executed.

- `:init-data {word}+`
  Put some initial data into the HAS-DATA slot of the Q-CONTEXT

- `:no-resulr-wait`
  If there, tell the dialog-crawler function not to wait for a user input.

- `:reset-conversation`
  Reset does several things:

  - clean the DATA and RESULT area in Q-CONTEXT
  - puts a new goal (action pattern) into the GOAL slot of conversation

- `:send-message`
  Sends a message to the PA, of type :internal, with action :send-request, with the structure of the message to be sent by the PA. This is used to ask for help or more generally to send a message during the course of a dialog.

Analysis of the user input.

The idea here is to analyze the input (from master or from another agent) and to determine if we should make a transition to another state. Hence the name "*immediate transitions*." When no conclusion can be reached, we can ask the user or another agent, wait for the answer and let the state-crawler function call the =resume method.

- `:immediate-transitions`
  The options associated with the immediate transition mechanism are normally quite simple and allow specifying simple processing on the data. When it is not possible to process the data finely enough, various escape mechanisms are provided, the ultimate one being to write the =execute method manually.

  The following options are simple rules that are applied to the content of the HAS-DATA slot of the context object (in the data variable). First MOSS looks for keywords in the data input. If one of the following is present: `:quit :abort :exit :reset :cancel` a transition is done onto the _Q-abort global state (the same state for all conversations).

  - Rules have premisses:

    `:always`
    fires immediately.

    `:contains {word}+`
    fires if one of the words appears in the data

    `:empty`
    tests if the DATA is empty

    `:no`
    fires when data contain "no N nein non never nimmer jamais"

    `:otherwise`
    at the end of the list, fires as a last resource

    `:patterns`
    we got a set of patterns ELIZA style. Apply the process-pattern function to each one.

    `:special-test :method <test method> :args <arg list> :res-var <var name>`
    applies a special test method of the conversation object to the data and sets the internal variable specified by the `:res-var` parameter (usually DATA or RESULTS).

`:starts-with {word}+`
: fires when the data start with one of the words

`:test expr`
: fires if the test applies (expr evaluates to non nil)

`:yes`
: fires when data contain ”yes Y oui Ja Jawohl”

- Rules have conclusions that should end up on a transition:

`:ask`
: activates the conversation pane (?)

`:clean-data <arg list>`
: sends itself a =clean-data method with conversation as argument followed by arg-list.

`:ELIZA`
: calls the ELIZA set of rules to further analyze data.

`:erase`
: erases the data.

`:exec <lisp expr>`
: executes lisp-expr

`:failure <arg list>`
: returns immediately with (:failure).

`:keep`
: replaces the content of DATA, saving it into the HAS-DATA conversation slot

`:save <arg list>`
: sets the data to arg-list

`:self`
: sends itself a method with conversation and arg list as parameters.

`:set-context <arg list>`
: send a =replace method with the arg list to replace whatever is mentioned.

`:set-performative <arg list>`
: sets HAS-PERFORMATIVE slot of the conversation to arg-list.

`:set-results <arg list>`
: sets HAS-RESULTS slot of the conversation to arg-list.

`:replace <arg list>`
: replaces the data with arg-list.

`:reset <arg list>`
: throws to a :dialog-error tag.

`:sub-dialog <arg list>`
: returns with a list (:sub-dialog <arg list>).

`:success <arg list>`
: returns immediately with (:success).

`:target <arg list>`
: returns with a list (:transition <arg list>).

`:trim-data {word}+`
: removes the list of words from the data

Most of the simple options may be replaced by more general patterns.
The last line of the =execute method is a call to the =resume method.

7.1.2 The =resume method

Called after a user input. The user data are in the TO-DO slot of the current conversation and are transferred into the INPUT lexical variable. The TO-DO slot is reset. Whenever the answer contains ”`:quit :abort :exit :reset :cancel`” a transition to the global _Q-abort state occurs.

Options are the following.

- `:answer-analysis`
  Indicates that a special method named =answer-analysis will be applied to the data (input vairiable). The method must return a transition state or the `:abort` keyword, in which case the conversation is locally restarted.

- `:execute function args`
  Executes the corresponding function and args

- `:input-from-data`
  The input is taken from the HAS-DATA slot of the conversation object rather than from the HAS-TO-DO slot in which an answer is returned.

- `:resume-action <method-name> <arg list>`
  Sends a message to the state object including the method and arguments.

- `:resume-var {DATA|RESULTS|CONTEXT|variable-name}`
  Allows to specify lexical (local) variables that will be used when executing the result of the macro expansion, i.e.in the synthetized =resume method.

  - DATA is initialized with the content of the HAS-DATA slot of the Q-CONTEXT
  - CONTEXT is initialized to the id of the Q-CONTEXT object
  - RESULTS is initialized to the content of the HAS-RESULTS slot of Q-CONTEXT
  - variable-name is any name chosen by the user

Analysis of the master answer.

- `:transitions`
  Similar to the :immediate-transitions of the =execute method. Options are:

  - Rules premisses:

    `:always`
    Always fires (should be last clause)

    `:contains {word}+`
    fires if one of the words appears in the data

    `:empty`
    fires when INPUT is empty (NIL)

    `:no`
    fires when data contain ”no N nein non never nimmer jamais”

    `:otherwise`
    always fire (should be the last option)

    `:patterns`
    we got a set of patterns ELIZA style and apply the process-pattern function to each one

    `:starts-with {word}+`
    fires when the data start with one of the words

    `:test <Lisp expr>`
    executes the Lisp expr and fires if the result is not null

    `:yes`
    fires when data contain ”yes Y oui Ja Jawohl”

  - Rules conclusions:

    `:clean-data args`
    sends itself a =clean-data method with conversation as the firs argument followed by the specified arguments and replaces the content of HAS-DATA with the value of input

`:exec <expr>`
  executes the Lisp expr

`:failure`
  returns immediately the list (:failure)

`:keep <arg list>`
  keep the value of the specified data in HAS-DATA

`:replace <arg list>`
  replaces the content of HAS-DATA with arg list

`:reset`
  resarts the local (sub-)conversation. Default is to restart the conversation at the beginning.

`:save`
  save the content of INPUT into the HAS-DATA field of Q-CONTEXT

`:self method args`
  sends itself the specified methof with conversation as the firs argument followed by the specified arguments

`:set-performative <arg list>`
  set the HAS-PERFORMATIVE slot to the specified value

`:sub-dialog dialog-header`
  specifies a sub-dialog

  – :success state : transfer state in case of success
  – :failure state : transfer state in case of failure

`:success`
  returns immediately the list (:success)

`:target state`
  specifies the transition state

`:trim-data`
  removes the words specified in the :contains or starts-with option from the value of data. Updates HAS-DATA

7.2 Defining the =execute Method Manually

The =execute method is normally synthesized by the `defstate` macro. However, the method can be specified manually. Its input parameter is a conversation object, it should end with a transition. An example of a manually specified =execute method is the following:

```
(m-defownmethod
 =execute _hc-display-results (conversation)
 "save the obj-id of the result into the action pattern and executes it."
 (let* ((state-context (car (HAS-Q-CONTEXT conversation)))
        (pattern (car (HAS-GOAL conversation)))
        (obj-arg (car (HAS-operator-arguments pattern)))
        (obj-id (car (HAS-RESULTS state-context)))
        )
   ;; add value to the arg
   (send obj-arg '=replace 'HAS-ARG-VALUE (list obj-id))
   ;;execute pattern
   (send pattern '=activate)
   ;; indicate success in results
   (setf (HAS-RESULTS state-context) '(:success))
   ;; transition to success
   (send *self* '=set-transition conversation _hc-success)
   ))
```

7.3 Defining the =resume Method Manually

The =resume method is used to analyze the user's answer after a question has been asked at a particular step of the dialogue. It is normally synthesized by the `defstate` macro but can be specified by the user directly.

7.4 Example of Miscellaneous User-Controlled Methods

Miscellaneous methods can be written to help the analysis. For example a specific =test method. However, those methods are part of an advanced used of the dialog interface.

```
(m-defownmethod
 =test _pc-entry-state (conversation)
 "Entry method, checks for an entry point. If there, go to ACCESS-EP.
Arguments:
   conversation: current conversation"
 (let* ((state-context (car (HAS-Q-CONTEXT conversation)))
        (input (car (HAS-DATA state-context)))
        (input-length (length input))
        (make-entry-method-list (send '=make-entry '=get-id '$MNAM.OF))
        entry-point-list prop-list ep expr)
   ;; cook up a list of properties corresponding to the list of the =make-entry
   ;; methods (there is at least one)
   (setq prop-list
         (mapcar #'car (broadcast make-entry-method-list '=get-id '$OMS.OF)))
   ;; try the whole data as a possible entry point (should be all atoms)
   ;; if input contains strings everything must be a string
   (when (or (every #'stringp input)
             (not (some #'stringp input)))
     (dolist (prop prop-list)
       (setq ep (-> prop '=make-entry input))
       (verbose-format "~&...trying ~S as an entry-point for ~S...ep? ~S"
                       ep prop (%entry? ep))
       (if (%entry? ep) (pushnew ep entry-point-list))))
   ;; if input has more than one element, try each element in turn
   (when (cdr input)
     ;; look for a word that could be an entry point
     (dolist (word input)
       ;; consider now every property that has entry-points in turn
       (dolist (prop prop-list)
         ;; try to make an ep
         (setq ep (-> prop '=make-entry (list word)))
         (verbose-format "~&...trying ~S as an entry-point for ~S..." ep prop)
         (if (%entry? ep) (pushnew ep entry-point-list)))))

   ;; if we did not find anything and input has several elements try them
   ;; several at a time (this could waste some symbol space)
   (unless entry-point-list
     (dotimes (nn (1- input-length))
       ;; one loop for pairs, triplets, etc
       (dotimes (jj (- input-length nn 2))
         ;; extract pairs, triplets, etc.
         (setq expr (subseq input jj (+ jj nn 2)))
         ;(print expr)
         (dolist (prop prop-list)
           ;; try to make an ep
           (setq ep (-> prop '=make-entry expr))
           (verbose-format "~&...trying ~S as an entry-pointfor ~S..." ep prop)
           (if (%entry? ep) (pushnew ep entry-point-list))))))
   ;; return result
   entry-point-list))
```

References

[Nor92] Peter Norvig. *Artificial Intelligence Programming*. Morgan Kaufmann, San Mateo, CA, USA, 1992.