UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique
UMR CNRS 7523 Heudiasyc

MOSS 8 - User Manual
Issue 1

Jean-Paul Barthès

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

Email: barthes@utc.fr

N255
April 2013

Warning

This document contains the documentation for the MOSS v8 Knowledge Representation environment.

The current research version of MOSS 8 runs in Allegro Common Lisp (ACL 9.0 running under Windows XP or Windows 7). The document gathers all the various documents previously available for MOSS v7 upgraded to correspond to version 8.

Keywords

Multi-agent platform, cognitive agents, OMAS.

Revisions

Version	Date	Author	Remarks
0.1	Apr 2013	Barthès	Draft

Contents

Introduction

History

The MOSS system is a prototype object environment designed and developed at UTC by Jean-Paul Barthès, starting in 1986. As a research system its main goal was to study the various object mechanisms proposed in the literature, to evaluate their relevance and efficiency, so as to keep the best ones and to design new ones. MOSS draws on past experiences with semantic representations, and object oriented systems. Objects use the PDM4 format (PDM stands for Property Driven Model).

MOSS was designed to be an homogeneous environment with great flexibility, the main goal being to support multi-user interactive AI applications. The approach is built upon a simple but powerful object representation around which an object-oriented language, and a multi-user object manager for storing data permanently on secondary storage. The first overall MOSS architecture is shown on Fig.1.

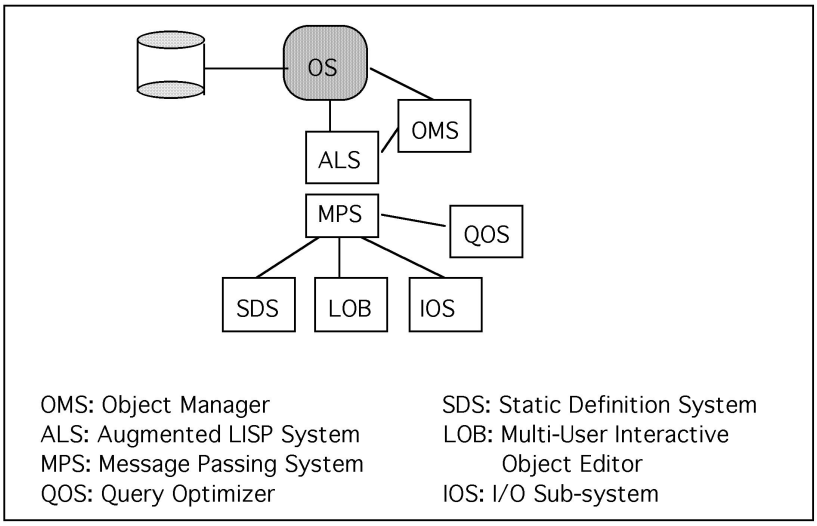

Figure 1: Overall architecture of the MOSS system

MOSS Today

MOSS can be used in standalone for representing knowledge through the development of ontologies and associated knowledge bases. MOSS can also be used in combination. For example, it has been integrated into the OMAS multi-agent platform for describing and implementing each agent knowledge. A more specialized version named SOL (Simple Ontology Language) can be used to translate an ontology into the OWL format.

The current MOSS architecture for version 8 contains only the object manager OMS and the message passing system MPS. The I/O system is that of Lisp. An interactive editor MES is now available.

Who should Read the Manual

This manual is intended for those who would like to get familiar with MOSS by exercising the MOSS environment. It describes how to use the MPS (Message Passing System) and the API (Programming Interface), which is good enough for prototyping small applications and catching a glimpse of what it is all about.

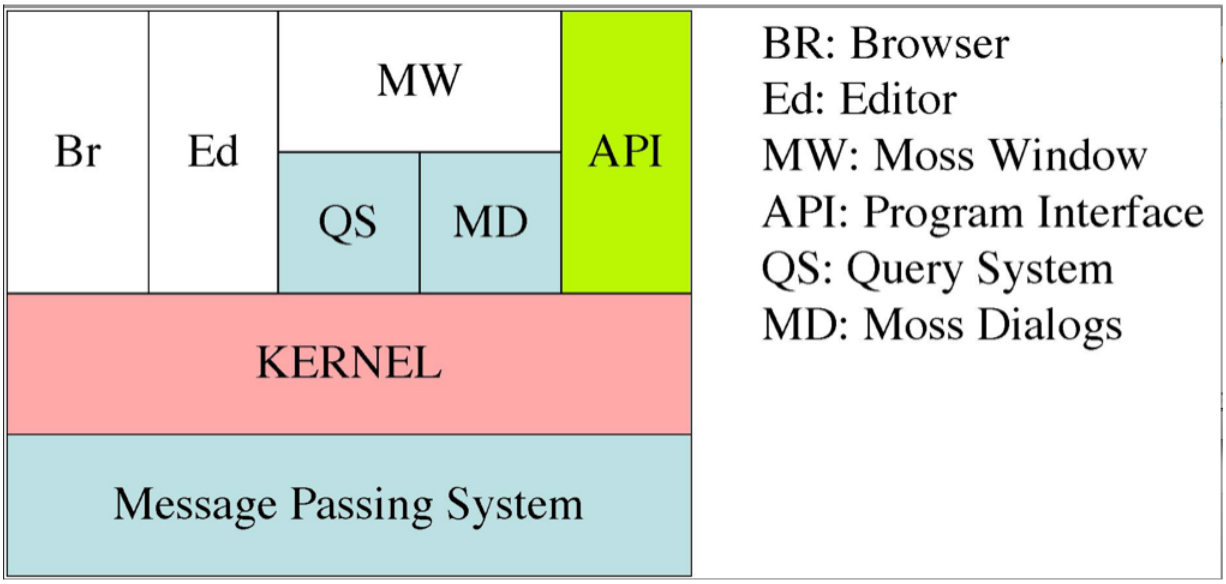

Figure 2: The MOSS components

Chapter 1

Primer

Contents

MOSS was designed to run in the Macintosh environment on top of MCL (Mac Common Lisp) and in the Windows environment on top of Franz Lisp ACL (Allegro Common Lisp®). However, when Apple switched to Intel processors, MCL was discarded and today only the Windows version is available.

Working with MOSS requires familiarity with the Lisp language.

1.1 Get Started

MOSS 8 was tested in ACL 9.0 environment running under Windows XP or Windows 7.

You start the ACL LISP and load the `moss-load.lisp` file, using the "Load File" entry of the File menu. Some messages appear like:

```
CG-USER(1):
; Fast loading C:\Program Files\Allegro CL 9.0\OMAS-MOSS 10.0.2\MOSS\moss-load.fasl
```

```
#P"C:\\Program Files\\Allegro CL 9.0\\OMAS-MOSS 10.0.2\\MOSS\\MOSS-version-info.lisp"
;   Loading C:\Program Files\Allegro CL 9.0\OMAS-MOSS 10.0.2\MOSS\MOSS-version-info.lisp
;   Fast loading C:\Program Files\Allegro CL 9.0\OMAS-MOSS 10.0.2\MOSS\MOSS source\FASL\moss-m

;*** MOSS v8.0.0 - Macros loaded ***
;   Fast loading
;      C:\Program Files\Allegro CL 9.0\OMAS-MOSS 10.0.2\MOSS\MOSS source\FASL\moss-service.fas

;*** MOSS v8.0.0 - Service loaded ***
;   Fast loading C:\Program Files\Allegro CL 9.0\OMAS-MOSS 10.0.2\MOSS\MOSS source\FASL\moss-b

... <more>

;*** MOSS v8.0.0 - Patches loaded ***
;   Fast loading C:\Program Files\Allegro CL 9.0\OMAS-MOSS 10.0.2\MOSS\MOSS source\FASL\moss-d

;*** MOSS v8.0.0 - Moss dialogs loaded ***
;   Fast loading C:\Program Files\Allegro CL 9.0\OMAS-MOSS 10.0.2\MOSS\MOSS source\FASL\moss-e

;*** MOSS v8.0.0 - Export functions loaded ***
;   Fast loading
;      C:\Program Files\Allegro CL 9.0\OMAS-MOSS 10.0.2\MOSS\MOSS source\FASL\moss-online-doc.

;*** MOSS v8.0.0 - online doc loaded (control) ***
;*** MOSS version 8.0.0 loaded ***
;*** Methods are cached: *cache-methods* set to T ***
;*** Fancy inheritance turned off *lexicographic-inheritance* set to T ***
CG-USER(1):
```

The messages should not bother you. Then an initial window appears (Fig.1.1).

Figure 1.1: MOSS start window

Clicking the START button calls the command window (Fig.1.2).

The MOSS command window contains a number of buttons referring to possible options. Because we need an initial environment for building an ontology, make sure that the ontology slot contains the name STARTER, which will provide the initial environment and the package slot should contains a question mark or nothing. Then click the IMPORT/LOAD button to load the starter environment.

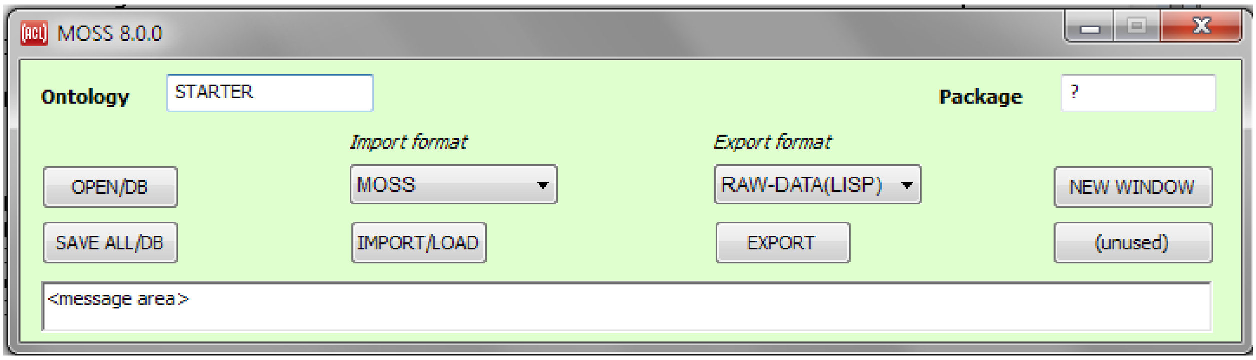

Figure 1.2: MOSS command window

Doing so calls makes a new window to appear (Fig.1.3).

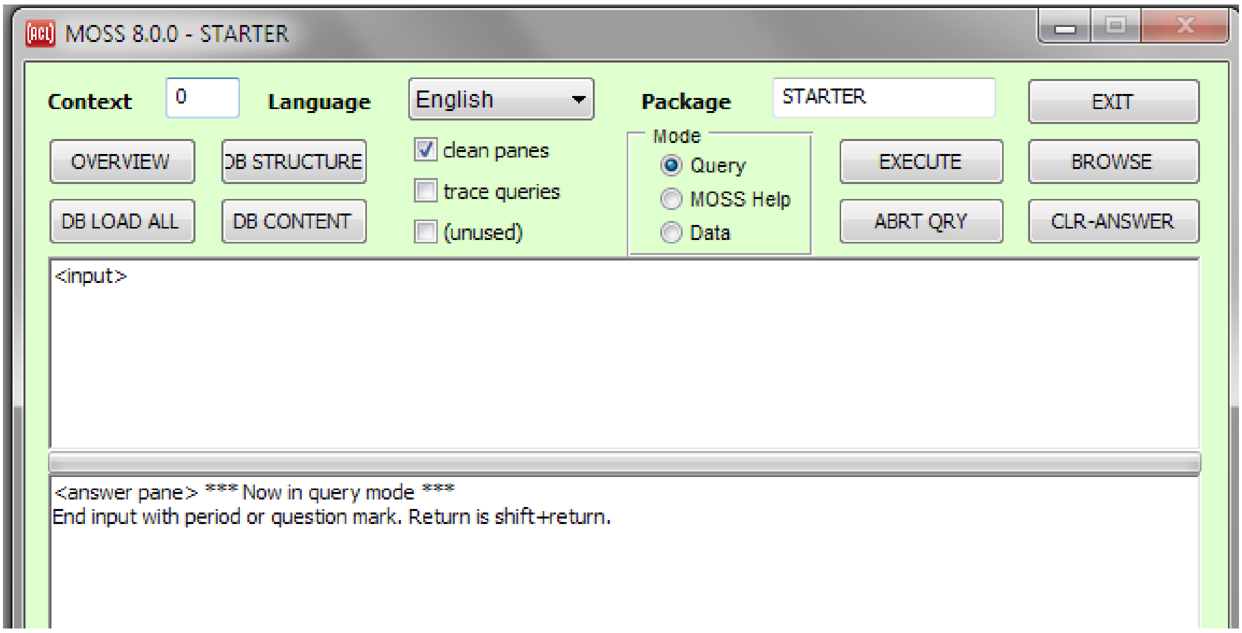

Figure 1.3: MOSS window

This window allows working with MOSS when dealing with large ontologies. However, on a Primer we want to test simple commands, thus we can safely discard this window by clicking the EXIT button and turn our attention to the ACL Debug Window.

1.2 Working with MOSS

Working with MOSS consists in creating objects and giving them behaviors. MOSS being a representation language allows in particular modeling ontologies. Thus, one tends to use concepts rather then classes and individual concepts rather then instances. Moreover, using concepts and individuals rather than classes and instances also avoids possible conflicts with CLOS (the internal Common Lisp Object System). Thus, we will use mostly the terms concept and individual rather than class and instance.

When you work in the ACL debug window prompts look like:

```
CG-USER(2):
```

meaning that you are in the CG-USER package and this is the second expression that you want the system to evaluate.

In order to switch to our test package type the following line:

```
CG-USER(3): :pa starter
```

which switches to the STARTER package (the :pa command stands for in-package). This is shown as:

```
STARTER(4):
```

1.2.1 Creating a Concept

Creating a concept is simple and can be done with the `defconcept` macro as follows:

```
STARTER(4): (defconcept "Human")
$E-HUMAN
```

This creates an object whose concept name is HUMAN in the STARTER environment. The macro returns `$E-HUMAN`, the internal name or identifier (**object-id**) referring to the concept you have just created. This identifier will be used as a radix when you create individuals, i.e., the individual internal names will be `$E-HUMAN.1`, `$E-HUMAN.2`, `$E-HUMAN.3`, and so on. A concept has an associated internal counter for building such names that starts counting at 1.

Internal and External Names: Object-ids and Entry Points

Objects in MOSS are unique entities. As such they exist, even if nothing is known about them, which is quite different from the relational model for example, in which an object (tuple) cannot exist if one knows nothing about it. The internal or system name for an object is called its identifier or **object-id**. An object identifier is unique and designates the corresponding object directly. However, such names are often meaningless and difficult to use. Furthermore, they are assigned by the system dynamically and depend on the order in which defining files are loaded. Consequently, as a facility to the user, external names can be specified. External names are called **entry points**, and are in practice indexes for accessing corresponding objects directly[1]. They are somewhat analogous to keys in traditional databases; but, contrary to keys, *they need not be unique*. Some of the defining macros provide external names automatically in a normalized way. Examples will be given as we go along. Recovering an internal name from an external entry point is sometimes not easy, due to possible ambiguities. Furthermore, for certain types of objects (e.g. concept names), external names must be unique within the application name space.

In the example, `$E-PERSON`, is the internal name or object-id of the concept PERSON. The concept PERSON is a first class object [2].

The symbol `PERSON` on the other hand is defined as the external name or entry point for the concept. *It is a reserved name and should not be used as the name of a special (global) variable, otherwise the structure of your application will break.*

[1]In practice the object-id is implemented as a LISP symbol, the value of which is an a-list representing the object itself. Hence one can view the internal structure of a MOSS object simply by typing its object-id.

STARTER(4): $E-HUMAN
((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-HUMAN)) (MOSS::$ENAM (0 (:EN "Human")))
(MOSS::$RDX (0 $E-HUMAN)) (MOSS::$ENLS.OF (0 $SYS.1)) (MOSS::$CTRS (0 $E-HUMAN.CTR)))

[2]Concepts, properties (attributes and relations), entry points are all first class objects

Cases

Because MOSS is defined in a traditional Lisp environment, cases are not important when using symbols and are not retained by the system, except within strings. Thus:

```
STARTER(4):  (defconcept HUMAN)
STARTER(4):  (defconcept Human)
STARTER(4):  (defconcept huMAN)
STARTER(4):  (defconcept HuMan)
```

will give the same result and the name of the concept will be printed as HUMAN, using capital letters. If one wants to retain the case [3], the definition using the string will keep the case when printing the external name of the concepts. It is advised to use the string version of the definition starting concept names with a capital letter, which is a somewhat standard approach for class names in OO languages.

A composed class name like "Capital Town" will yield the normalized external name CAPITAL-TOWN. Note the hyphenation mark that is not an underscore.

Documentation

One can also attach some documentation information to a concept by using the option `:doc`, e.g.

```
STARTER(4):  (defconcept "HUMAN"
                (:doc "This describes a HUMAN, bla, bla,..."))
```

Redefining Concepts

Note also that if you want to redefine the concept HUMAN using the defconcept macro, then MOSS issues an error. Indeed it is an error to redefine a concept. Thus, if you want to test a different definition without reinitializing your session, you have to execute it with a different concept name (e.g. "Human-1").

From now on, you could create individuals of the concept HUMAN using this model. However, since the concept has no attributes, this is not very interesting. Next section indicates how to add properties to concepts.

1.2.2 Creating Concept Properties

MOSS objects have two kinds of properties: immediate values that qualify the object, and links to other objects. The first kind is called an **attribute**, the second kind is called a relationship or **relation**. Many people use the terms attribute, property, and relation to mean quite different things. We will use them strictly as shown in the next paragraphs.

Attributes

Within the `defconcept` macro, you can specify that a PERSON has a name, and a first name with the `:att` (attribute) option.

```
STARTER(5):  (defconcept "Person"
                (:att "name" (:min 1)(:max 3)(:entry))
                (:att "first name"))
```

[3] This is only useful when printing

Because *attributes can be multivalued*, you may want to specify the minimal and the maximal cardinality. In the above example a PERSON will have at least 1 name and at most 3. The `:entry` option means that you want to be able to get access to the persons (instances) by giving their name (in case of multiple names, any one of them will do). Hence the value of the name property for an instance will generate one or more **entry point(s)**.
For the first name, no cardinality restriction is imposed: a person can have zero or an unlimited number of first names.
Notice also that no specific type has been imposed on the attributes.

Creating Attributes Directly It is important to realize that *properties can be created independently from any concept.* For example:

```
STARTER(5):  (defattribute "sex")
$T-SEX
```

defines a single-valued generic property [4]. The system returns the internal name, e.g. $T-SEX, of the property.

Instead, the attribute can however be attached to a concept as follows:

```
STARTER(5):  (defattribute "sex" (:min 1) (:max 1) (:concept PERSON))
$T-PERSON-SEX
```

Note that the internal name is tagged with the concept name. The attribute is defined as a local attribute related to the concept PERSON, rather than a global attribute that exists independently from any concept. The external name of the attribute in both cases is constructed by adding the prefix HAS: `HAS-NAME`.

Relations

Relations, also known as links, can be defined likewise, using the `defrelation` macro:

```
STARTER(6): (defrelation "brother" (:from "Person")(:to "Person"))
$S-PERSON-BROTHER
```

meaning that a person may have another person as a brother. The actual meaning of the notation is more like that of an integrity constraint: PERSON appears twice: the first time to specify the concept it is a property of (sometimes called the domain e.g. for a function), the second time to specify the type of the "successor" (sometimes called its range).

Creating Relations Directly It is important to realize that *properties can be created independently from any concept.* For example:

```
STARTER(7): (defrelation "neighbor" (:from :any)(:to :any))
$S-NEIGHBOR
```

creates a relation object meaning that any object can be the neighbor of any other object. However, in general relations are defined between individuals (i.e. instances of concepts).

———-

In order to beef up our example we now define additional concepts:

[4] `defattribute` does not require that the created property be attached to a specific concept. It is thus possible to create properties independently (like mixins in the flavor system).

```
STARTER(8): (defconcept "Department"  (:att "name"))
$E-DEPARTMENT

STARTER(8): (defconcept "Course"
              (:att "code" (:min 1)(:max 3)(:entry))
              (:rel "department" (:to "Department")))
$E-COURSE
```

Note that, while defining the concept COURSE, we used DEPARTMENT with two meanings in the last line: one for a relation, the second one for the concept name. This situation happens quite frequently, hence MOSS creates distinct external names for concepts and for properties, removing the ambiguity. Here `DEPARTMENT` will be the external name of the concept, and `HAS-DEPARTMENT` the external name of the property.

1.2.3 Defining Sub-Concepts

Like in other representation languages, one can define sub-concepts as refinements of concepts. For example:

```
STARTER(9): (defconcept "Student"
              (:is-a "Person")
              (:rel "course" (:to "course")))
$E-STUDENT
```

or

```
STARTER(10): (defconcept "Teacher"
               (:is-a "person")
               (:rel "lecture" (:to "course"))
               (:rel "department" (:to "department")))
$E-TEACHER
```

As expected, students or teachers will inherit the properties of persons.

1.2.4 Creating Individual Concepts

Concepts are used to create individual concepts that will respect the structure specified by the concept. To create individuals one uses the `defindividual` macro.

```
STARTER(11): (defindividual "person")
```

will return the internal name of the instance, e.g. the first time you will get

```
$E-PERSON.1
```

To store the internal name in a variable, one can do a regular assignment or else use the `:var` keyword option, i.e.

```
STARTER(12): (setq _p1 (defindividual PERSON))
or
STARTER(12): (defindividual PERSON (:var _p1))
```

Note that by convention a variable name should start with an underscore, e.g., _P1. Note also that we can define individuals by referring to the class by means of the corresponding string [5] ("person") or by using the internal name (PERSON).

Of course, it is interesting to assign values to properties of individuals. This is done by using the external names of the properties in the `defindividual` macro.

```
STARTER(13): (defindividual PERSON
                (:var _p1)
                (has-name "Dupond" "Durand")
                (has-first-name "Jean")
                (has-sex "M"))
```

or

```
STARTER(14): (defindividual PERSON
               (:var _p2)
               ("name" "Dubois" "Dupond")
               ("sex" "F")
               ("brother" _p1))
```

Here the internal object-ids are kept in the variables _p1 and _p2. Property names can be specified by their definitional name prefixed with HAS-. Remember, this is to distinguish attributes from concepts: `HAS-DEPARTMENT` indicates a property and `DEPARTMENT` a concept name. Values can be multiple like the 2 names: Dupond/Durand or Dubois/Dupond.

Note that the `HAS-XXX` symbols can be replaced with the string `"XXX"` as shown in the second definition.

1.2.5 Printing Information

While creating objects it may be interesting to print their content. This is done simply by sending a message to whatever object must be printed in the following way:

```
STARTER(15): (send '$E-person '=print-self)
```

or

```
STARTER(16): (send _person '=print-self)
```

Notice the "=" sign in front of the `=print-self` method [6]. We obtain:

```
----- $E-PERSON
 CONCEPT-NAME: Person
 RADIX: $E-PERSON
 ATTRIBUTE : name/Person, first name/Person
 RELATION : brother/Person
 COUNTER: 4
-----
:DONE
```

[5] The case here is not important. We can use "name" or "Name" or "NAME".

[6] In some systems method names start with ":". In SMALLTALK they end with ":". In MOSS I chose to use the "=" sign to distinguish methods from other names (e.g. entry point names), and because the ":" sign in LISP is reserved for keywords and packages.

Note that the symbol `_person` (variable) holds the internal object-id of the concept PERSON and was defined by MOSS automatically. However, if we write

```
STARTER(17): (send PERSON '=print-self)
```

then we get an error. For printing the content of person `_p1` we write

```
STARTER(18): (send _p1 '=print-self)
```

getting

```
----- $E-PERSON.3
 name: Dupond, Durand
 first name: Jean
 sex: M
-----
:done
```

Similarly, for printing the content of the relation brother

```
STARTER(19): (send _has-brother '=print-self)
```

we get (generic brother relation)

```
----- $S-BROTHER
 INVERSE: (EN IS-BROTHER-OF)
 SUCCESSOR: UNIVERSAL-CLASS
 PROPERTY-NAME: brother
-----
:done
```

Note however that we printed here the generic brother relation and not the brother relation attached to the PERSON concept. To do so we must send a message as follows:

```
STARTER(19): (send _has-person-brother '=print-self)
```

which produces

```
----- $S-PERSON-BROTHER
 MOSS-INVERSE: (EN IS-BROTHER-OF)
 MOSS-SUCCESSOR: Person
 MOSS-PROPERTY-NAME: brother
-----
:done
```

etc.

1.2.6 Object Behavior

In the previous sections, we used only definitional macros and predefined methods. The interesting part in an object oriented programming language is to define specific behaviors for various objects. MOSS uses three types of methods as presented in the next paragraphs.

General Behavior

Assume that we'd like the method `=print-self` to print first names and names of persons. Instead of a tabular format, then we can define a specific `=print-self` method for persons using the `definstmethod` macro, as:

```
STARTER(20): (definstmethod =print-self PERSON ()
                "Prints a list of first names and names"
              (format t "~&~{~A~^-~} ~{~A~^-~}" (HAS-FIRST-NAME) (HAS-NAME)))
```

where `(HAS-FIRST-NAME)` is a short-hand notation for

```
(send *self* '=get 'HAS-FIRST-NAME)
```

where `*self*` is bound to the object that just received the message;

Writing the above definition creates an object method that is attached to the concept PERSON, but applies to its individual instances. Hence it is called an **instance-method**. It is used as follows:

```
STARTER(21): (send _p1 '=print-self)
Jean Dupond-Durand
NIL
```

Using the macro `definstmethod`, one can create any kind of instance method.

Note also that by convention methods begin with an = sign. This is not required, however it is good practice and helps separating method names from other names like variable names, property names, concept names, etc. MOSS uses this convention for predefined method names. Method names should not start with the ":" character, because Lisp uses it to indicate a keyword. Keywords cannot have values, hence cannot be used as entry points for methods.

Exceptions, Own Methods

Let us assume that we want to print Ms. Dubois differently by omitting her first name and inserting Ms for example. We can define a specific method, called **own-method** that we attach to the individual instance of person _p2. We do that using the special macro `defownmethod`:

```
STARTER(22): (defownmethod =print-self _p2 ()
                "Prints the name with Ms. in front of it."
              (format t "~&Ms ~{~A~^-~}" (HAS-NAME)))
```

Note that the concept name argument has been replaced with a variable containing the object-id.

The specific method will be called in due time, i.e. when we state:

```
STARTER(23): (send _p2 '=print-self)
Ms Dubois-Dupond
NIL
```

which yields a different result from

```
STARTER(24): (send _p1 '=print-self)
Jean Dupond-Durand
NIL
```

The NIL value returned in each case is the value returned by the printing function. Traditionally Lisp printing functions always return nil.

Universal Methods

Some methods like `=get` or `=print-self` are predefined and apply to all objects. They are called **universal methods** and are not attached to any object in particular. They are used by default as explained in the next section.

Inheritance Mechanism

When an object receives a message, MOSS looks first for a local or own-method. If one is there, then it applies it. Otherwise, it looks for an instance method at the concept level. If none is found, then it tries to inherit some from super-concepts using a mechanism described in the section about advanced features. If none can be inherited, then it looks for a universal method. If none exists, then it quits.

1.2.7 Creating Orphans

Because we wanted to use MOSS in robotics we introduced the possibility of having *objects not specified by any concept* or **orphans**. An orphan is an object that does not have a concept but that may have any predefined property. It can be built using the `defobject` macro as follows.

```
STARTER(25): (defobject ("name" "Joe")(:var _ob1))
$ORPHAN.1
```

defines an orphan with name Joe; `_ob1` is a variable containing its internal object id.
Of course we can attach (own-)methods to such an object to specify its behavior.

```
(defownmethod <method-name> _ob1 <args> <doc><body>)
```

Prototyping Furthermore we can specify that an orphan behaves as another known object by using the keyword `:is-a`. E.g.,

```
STARTER(26): (defobject (:var _p4)("NAME" "George")(:is-a _p2))
```

We don't know exactly what George may be, except that it behaves like `_p2`. Methods will be inherited along the is-a link. `_p2` is thus a **prototype** for the object we just defined. Thus, if we write

```
STARTER(27): (send _p4 '=print-self)
Ms   George
NIL
```

We notice that the special method defined previously for the person `_p2` has been used for the orphan `_p4` that we just created.

1.3 Programming

1.3.1 Message Passing

As with any object-oriented programming language one builds an application by defining objects and associated methods implementing the object behavior. Programming is done by sending messages. The main function to use is **send** as shown previously. A **broadcast** function is also available for sending the same message to a set of objects. However, it uses the Lisp `mapcar` primitive and thus *is not a parallel primitive.*

1.3.2 Service Functions

The programmer can use any of the Common LISP primitives in the methods. In addition a number of service functions are available. Some of them are very general and should actually be LISP primitives, others are related to the structure of the objects (PDM). They are efficient, but should be used sparingly and are mentioned in a separate document.

1.3.3 Predefined Methods

There are several kinds of predefined methods. Two kinds are of interest here:

basic methods like `=set-id`, `=set`, etc., that temper with the physical structure of the object representation. Such methods should not be used directly, because they can destroy the overall PDM structure (logical consistency).

PDM or kernel methods are defined at a higher level. They include `=add`, `=delete`, `=print-value`, etc.; such methods are referenced in the MOSS 6 Kernel Manual and should be used whenever possible. They respect the PDM structure (logical integrity constraints). In addition `=check` methods are provided to verify if an object still follows the PDM format.

Among the universal methods, some are of interest:

=get-properties returns the list of all properties as specified by the object concept, including the inherited ones. The resulting list looks strange; it is the list of internal ids of the properties.

```
STARTER(28): (send _p2 '=get-properties)
($T-PERSON-NAME $T-PERSON-FIRST-NAME $T-PERSON-SEX $S-PERSON-BROTHER)
```

=print-methods prints all instance methods associated with a concept. For the concept person, we defined only one method: `=print-self`.

```
STARTER(29): (send _person '=print-methods)
LOCAL INSTANCE METHODS
======================

=PRINT-SELF
Prints a list of first names and names
:DONE
```

=what? prints a summary of the type of object which received the message. If it is a concept, then prints the contents of the DOCUMENTATION field if it was filled (`:doc` option).

```
STARTER(30): (send _person '=what?)

The object is an instance of CONCEPT
*Sorry, no specific documentation available*
T
STARTER(31): (send _p4 '=what?)

The object is an orphan (i.e. has no class).
... has prototypes (depth first)
$E-PERSON.3
```

```
*Sorry, no specific documentation available*
T
```

1.3.4 Writing a Method

Global Variables

Methods are written in Common LISP. Several global variables are accessible when the method is executing, i.e. they can be used within the method.

self contains the internal name of the object that just received the message.

args contains the list of arguments of incoming message.

sender contains the identity of the sender. By default the initial sender is `*user*`.

After a method is executed it returns control to the sender, eventually with a value. The returned values are available in the ***answer*** variable.

Redefining a method

A method is redefined by re-executing the `definstmethod` or `defownmethod` macro. This automatically cleans the environment.

1.3.5 Debugging Helps

When writing new methods, it is sometimes of interest to visualize what is going on by monitoring the traffic of messages. MOSS allows doing several things:

print all traffic: all messages and answers are printed in an indented fashion,

```
(trace-message) or (ton)
(untrace-message) or (toff)
```

trace specific methods:

```
(trace-method <method-id>)
(untrace-method <method-id>)
```

monitor the traffic to specific objects:

```
(trace-object <object-id>)
(untrace-object <object-id>)
```

Example of tracing messages: When tracing messages, we monitor all messages that are exchanged in the system.

```
STARTER(32): (trace-message)
T
STARTER(33): (send _p1 '=print-self)
1 from: *USER* to: $E-PERSON.2, request: =PRINT-SELF arg-list: NIL
Jean Dupond-Durand
 1 $E-PERSON.2 for: =PRINT-SELF, to: *USER* returns: NIL
NIL
```

Example of tracing objects: A more complex trace is the following:

```
STARTER(34): (send _ob1 '=print-self)
 1 from: NIL to: $ORPHAN.1, request: =PRINT-SELF arg-list: NIL
  2 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-PROPERTIES arg-list: NIL
  2 $ORPHAN.1 for: =GET-PROPERTIES, to: $ORPHAN.1 returns: (MOSS::$TYPE MOSS::$ID $T-NAME)
----- $ORPHAN.1
  2 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-ID arg-list: (MOSS::$TYPE)
  2 $ORPHAN.1 for: =GET-ID, to: $ORPHAN.1 returns: (*NONE*)
  2 from: $ORPHAN.1 to: $TYPE, request: =PRINT-VALUE arg-list: ((*NONE*))
 MOSS-TYPE: *NONE*
  2 $ORPHAN.1 for: =PRINT-VALUE, to: $TYPE returns: NIL
  2 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-ID arg-list: (MOSS::$ID)
  2 $ORPHAN.1 for: =GET-ID, to: $ORPHAN.1 returns: ($ORPHAN.1)
  2 from: $ORPHAN.1 to: $ID, request: =PRINT-VALUE arg-list: (($ORPHAN.1))
 MOSS-IDENTIFIER: $ORPHAN.1
  2 $ORPHAN.1 for: =PRINT-VALUE, to: $ID returns: NIL
  2 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-ID arg-list: ($T-NAME)
  2 $ORPHAN.1 for: =GET-ID, to: $ORPHAN.1 returns: ("Joe")
  2 from: $ORPHAN.1 to: $T-NAME, request: =PRINT-VALUE arg-list: (("Joe"))
 name: Joe
  2 $ORPHAN.1 for: =PRINT-VALUE, to: $T-NAME returns: NIL
-----
 1 NIL for: =PRINT-SELF, to: $ORPHAN.1 returns: :DONE
:DONE
```

When tracing an object, we only monitor messages addressed to this object. Notice the difference with the previous trace.

```
STARTER(35): (trace-object _ob1)
T
? (send _ob1 '=print-self)
 1 from: NIL to: $ORPHAN.1, request: =PRINT-SELF arg-list: NIL
  2 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-PROPERTIES arg-list: NIL
  2 $ORPHAN.1 for: =GET-PROPERTIES, to: $ORPHAN.1 returns: (MOSS::$TYPE MOSS::$ID $T-NAME)
----- $ORPHAN.1
  2 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-ID arg-list: (MOSS::$TYPE)
  2 $ORPHAN.1 for: =GET-ID, to: $ORPHAN.1 returns: (*NONE*)
 MOSS-TYPE: *NONE*
  2 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-ID arg-list: (MOSS::$ID)
  2 $ORPHAN.1 for: =GET-ID, to: $ORPHAN.1 returns: ($ORPHAN.1)
 MOSS-IDENTIFIER: $ORPHAN.1
  2 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-ID arg-list: ($T-NAME)
  2 $ORPHAN.1 for: =GET-ID, to: $ORPHAN.1 returns: ("Joe")
 name: Joe
-----
 1 NIL for: =PRINT-SELF, to: $ORPHAN.1 returns: :DONE
:DONE
```

Example of tracing methods: When tracing a method, we only monitor messages that refer to the method name. Notice the difference with the previous traces.

```
STARTER(36): (trace-method '=get-id)
T
? (send _ob1 '=print-self)

----- $ORPHAN.1
 1 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-ID arg-list: (MOSS::$TYPE)
 1 $ORPHAN.1 for: =GET-ID, to: $ORPHAN.1 returns: (*NONE*)
 MOSS-TYPE: *NONE*
 1 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-ID arg-list: (MOSS::$ID)
 1 $ORPHAN.1 for: =GET-ID, to: $ORPHAN.1 returns: ($ORPHAN.1)
 MOSS-IDENTIFIER: $ORPHAN.1
 1 from: $ORPHAN.1 to: $ORPHAN.1, request: =GET-ID arg-list: ($T-NAME)
 1 $ORPHAN.1 for: =GET-ID, to: $ORPHAN.1 returns: ("Joe")
 name: Joe
-----
:DONE
```

1.3.6 Errors

When typing macros one can be faced with syntax errors or errors due to some internal inconsistencies. It is possible to obtain the type of error by executing the offending expression surrounded by a catch function:

```
STARTER(37): (catch :error (defconcept "Human" (:is-a "Mammal")))
Warning: when defining class (:EN "Human") in package #<The STARTER package>:
class Mammal in the :is-a option is not (yet?) defined. Discarded.
```

1.3.7 Defining Test Files

When trying to get acquainted with MOSS it is a good idea to group all definitions in two separate files, one for defining the object static structures, the second one for defining their behavior (methods). They could be called e.g. `mytest-concepts.lisp`, and `mytest-methods.lisp`. They can be loaded after the MOSS environment. Thus, one does not need to retype everything all the time.

Note however that concepts can be loaded only once. Indeed, as a safety feature, MOSS refuses to redefine concepts. Hence when modifying the definition file, the environment has to be reloaded. On the other hand, methods can be redefined at any time, and one can use a separate file so that the file can be edited and reloaded any number of times. When the file becomes big, one can create a temporary file for debugging a particular method, and, once the method works satisfactorily, copy it into the larger file.

1.4 Advanced Features

They are described Chapter **??**.

Chapter 2

User Interface

Contents

Warning MOSS has a *versioning mechanism*, i.e. all objects can have different versions in the same environment. The current document is intended for beginning users who want to explore a particular knowledge base that is assumed to be in *version 0* (also referred to as *context 0*).

2.1 Introduction - Loading the Application

After having launched the Lisp environment, MOSS can be started as a stand alone application by loading the moss-load.fasl file (Fig.2.1).

When started as a stand alone application, MOSS opens with an interface window shown Fig.5.1. Throughout the document we refer to this user's interface as the **MOSS window**. Then the MOSS initial window appears as shown Fig.2.2.

Clicking the START button calls the MOSS COMMAND window (Fig. 2.3).

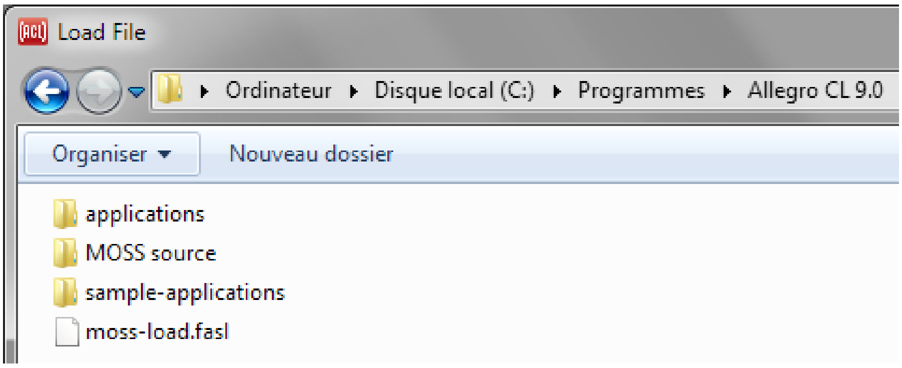

Figure 2.1: File for loading MOSS as a stand alone application: moss-load.fasl

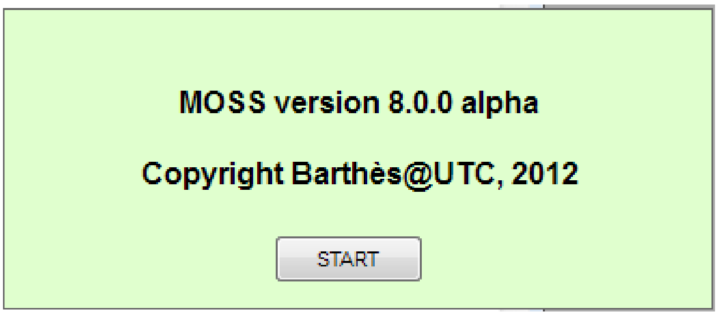

Figure 2.2: MOSS INIT window

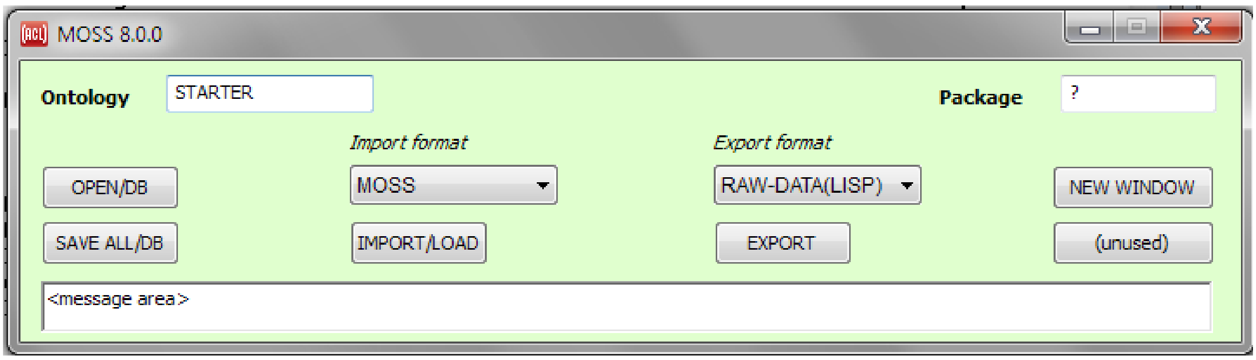

Figure 2.3: MOSS COMMAND window

This chapter deals with a predefined ontology, FAMILY. To load the ontology, one must type the name into the Ontology slot replacing whatever is there, and click the IMPORT/LOAD button. Doing so brings up the MOSS window (Fig. 5.1).

In addition to the MOSS window, other more specialized windows can be activated for locating objects, editing objects, or browsing through the knowledge base. Section 2.2 describes the MOSS command window, Section 2.3 introduces the concept display, Section 2.4 the browser, Section 2.5 the experimental help dialogs. The interactive editor is described in a separate document:

<Editor reference>

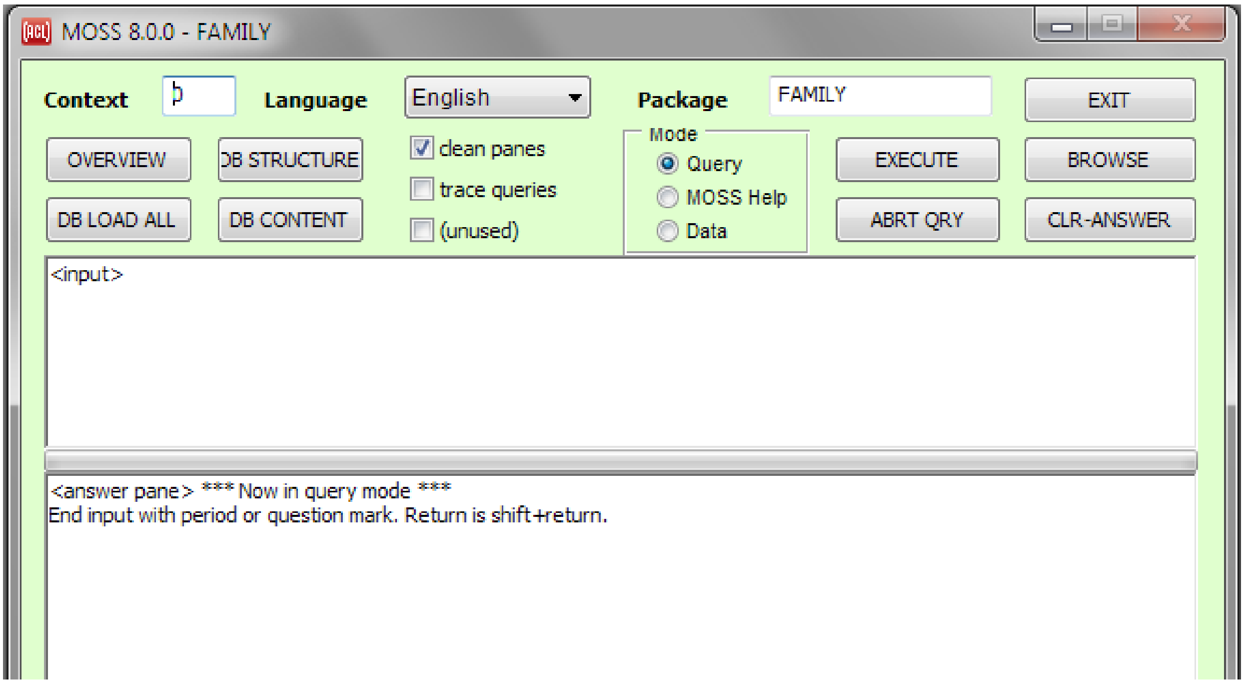

Figure 2.4: User Interface - MOSS window

2.2 User's Interface - MOSS Window

The MOSS window is essentially a support for executing MOSS queries on a given database. It allows to extract data and display it, possibly with details in BROWSER windows. It also allows editing objects from the BROWSER windows. However, for simply browsing through the ontology and the individuals, the CONCEPT DISPLAY window, detailed in Section 2.3 and called by clicking the OVERVIEW button, is easier to manipulate.

The MOSS Window has several areas: A top line, rows of buttons underneath, a first display area labeled <input>, a second display area labeled <answer pane>, and in between the two display areas a progress line (that shows in grey). This section introduces the various features of the MOSS window.

2.2.1 Exploring the Family World

Note The input pane is used to give commands. Ending the input depends on the mode we are in:

- in the query mode, inputs are queries and must obey the query format (refer to the document on MOSS queries). An important point is that there must be as many closing parentheses as opening ones. To execute the command, close enough parentheses and hit the return key.
- in the help mode, a command is a text that ends with a period or a question mark. Hitting the return key simply allows entering more text in continuation lines.
- in the data mode the user may type any Lisp expression, and hit the return key.

Simple Queries

Now, let us explore the family world. If we do not know what are the concepts of the FAMILY knowledge base, we can ask for them by using the formal request

```
("MOSS concept")
```

followed by carriage return. MOSS displays the available concepts in the ANSWER PANE (Fig.2.5). If we want to display results on a clean page each time, we check the clean panes box.

In the example, we obtain a list of four concepts: PERSON, COURSE, STUDENT, and ORGANISM. Note that during the access to the object base a progress line (green, in between the two large panes) shows the progression (this however is mostly for complex queries that take some time to execute).

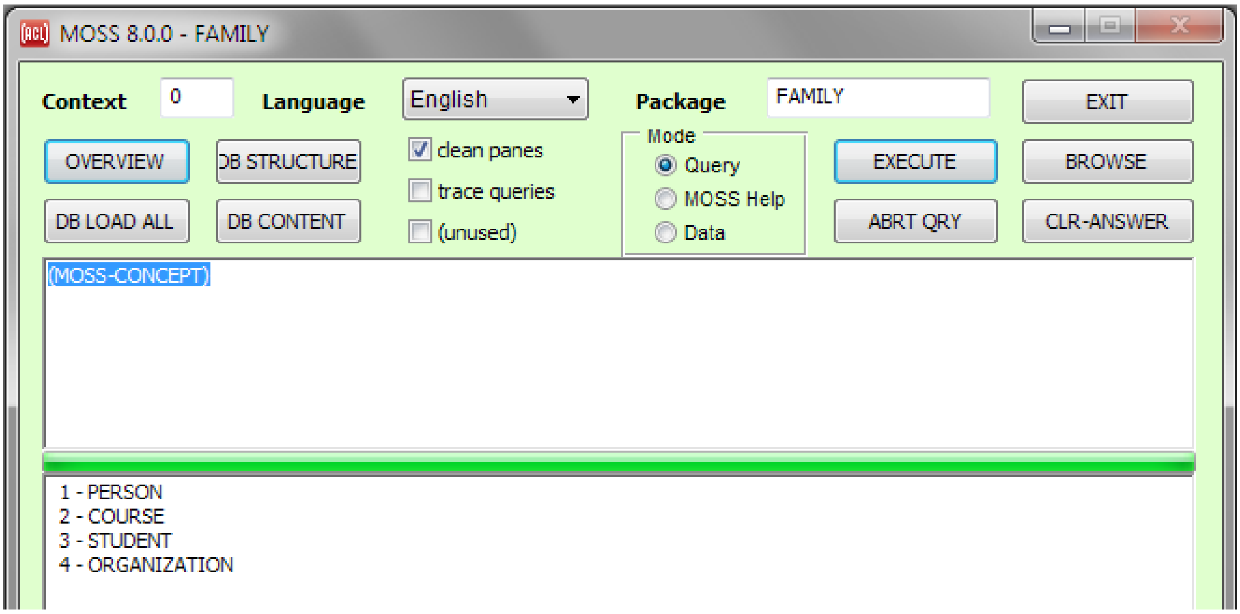

Figure 2.5: Concepts of the Family knowledge base

If we want now to obtain the list of individuals instances of person, we type

```
("person")
```

into the input pane. The result appears in the ANSWER PANE (Fig.2.6).

The objects are displayed using a =summary method attached to the concept of PERSON. In the example it prints the first name followed by the family name of a person.

More Complex Queries

More complex queries can be asked, reducing the list of results:

- Fig.2.7 for example gives the students whose name is "Barthès". The corresponding query is:

  ```
  ("student" ("name" :is "barthès"))
  ```

- Fig.2.8 gives the list of persons whose name is "Barthès" by using the entry point:

  ```
  "barthès"
  ```

  Note that the same result could be obtained by using the query:

  ```
  ("person"("name" :is "barthès"))
  ```

Figure 2.6: Person individuals in the Family knowledge base. Query: ("person")

Using the entry point (index) directly returns the list of all objects in the knowledge base for which "barthès" is en entry point. It may include objects that are not persons.

- Fig.2.9 gives the list of persons whose name is "Barthès" and who have a sister. The corresponding query is:

```
("person" ("name" :is "barthès")
          ("sister" ("person")))
```

- Fig.2.10 gives the list of persons whose name is "Barthès" and who have at least two brothers. The corresponding query is:

```
("person" ("name" is "barthès")("brother" (> 1) ("person")))
```

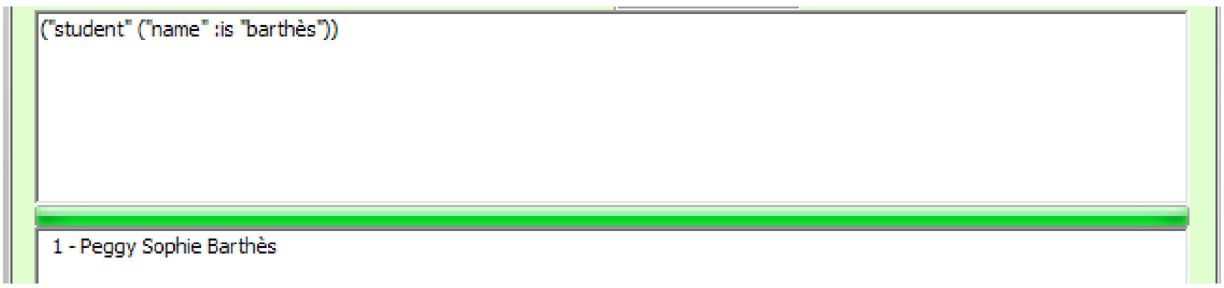

Figure 2.7: Query: ("student" ("name" is "barthes"))

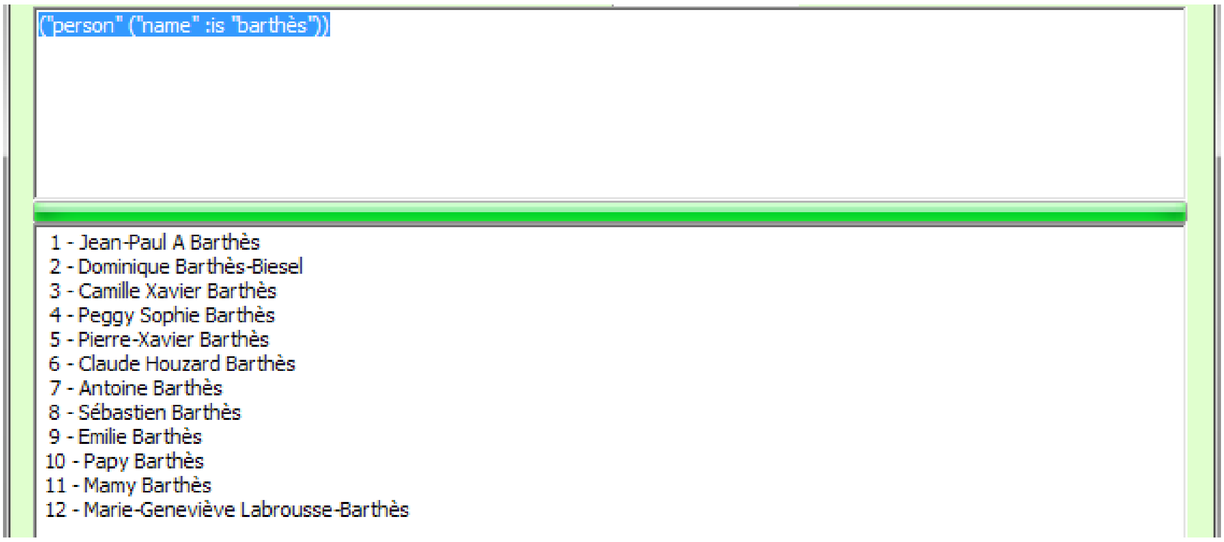

Figure 2.8: Query: "barthes"

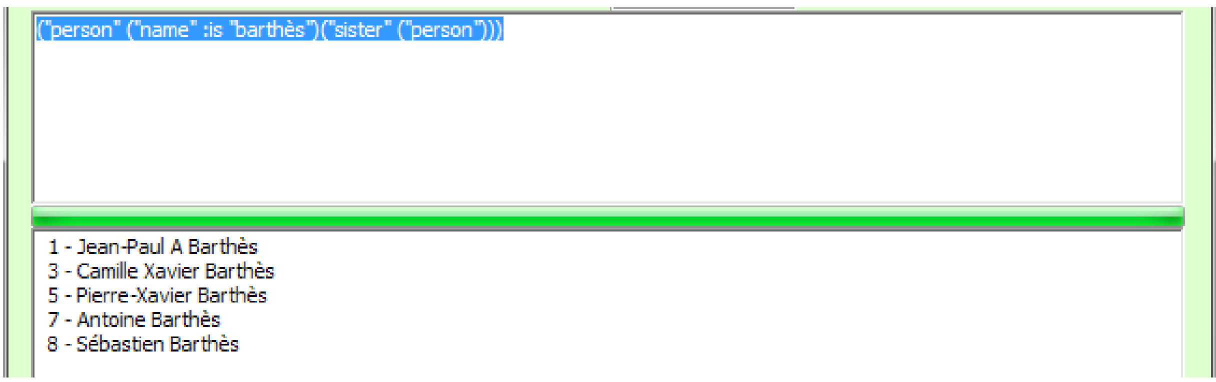

Figure 2.9: Query: ("person" ("name" is "barthes") ("sister" ("person")))

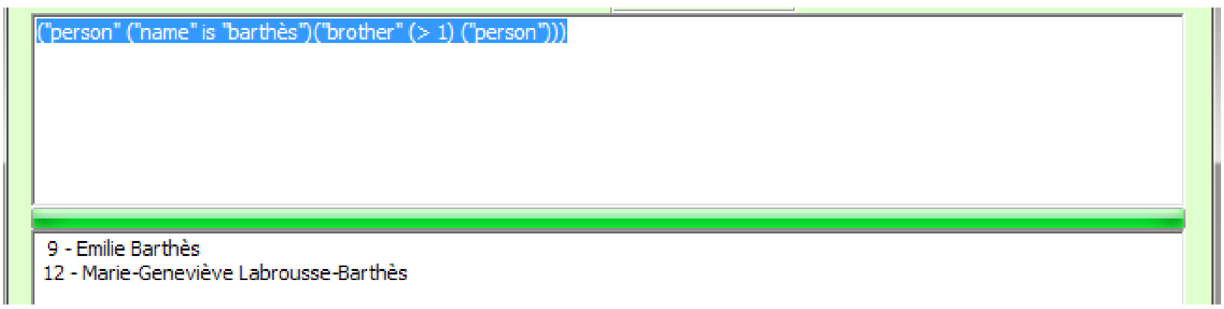

Figure 2.10: Query: ("person" ("name" is "barthès") ("brother" (> 1) ("person")))

The syntax for queries can be quite complex and is detailed in the chapter "Query System."

Note If a query takes too long, it can be aborted using the ABRT QRY button. The message "Query aborted at user request" appears in the ANSWER PANE.

2.2.2 Obtaining More Details about an Object

Objects listed in the answer area can be selected. For example, selecting the Emilie Barthès entry in the list of answers and clicking the BROWSE button makes a browsing window appear as shown Fig.2.11.

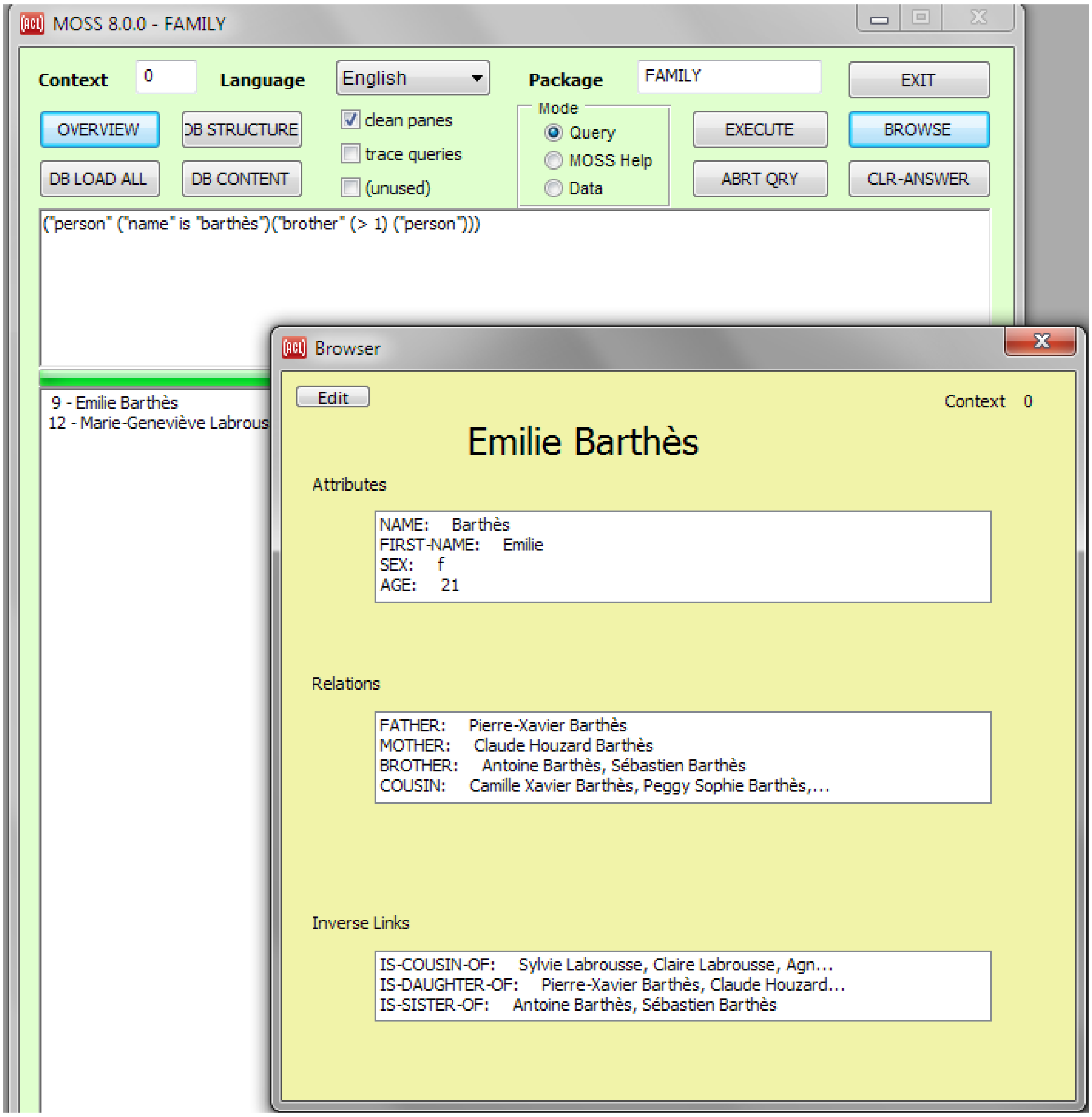

Figure 2.11: Detailing an object

Then, it is possible to browse the database from there (see Section 2.4), or to edit the object [1] by pushing the Edit button (Fig.2.12), or to discard the browser window by closing it.

2.2.3 Browsing Classes and Instances "Ontology" Style

Clicking the OVERVIEW button triggers the CONCEPT DISPLAY, which is described in Section 2.3.

[1] Note that the object will be edited in memory and not saved back into the original file

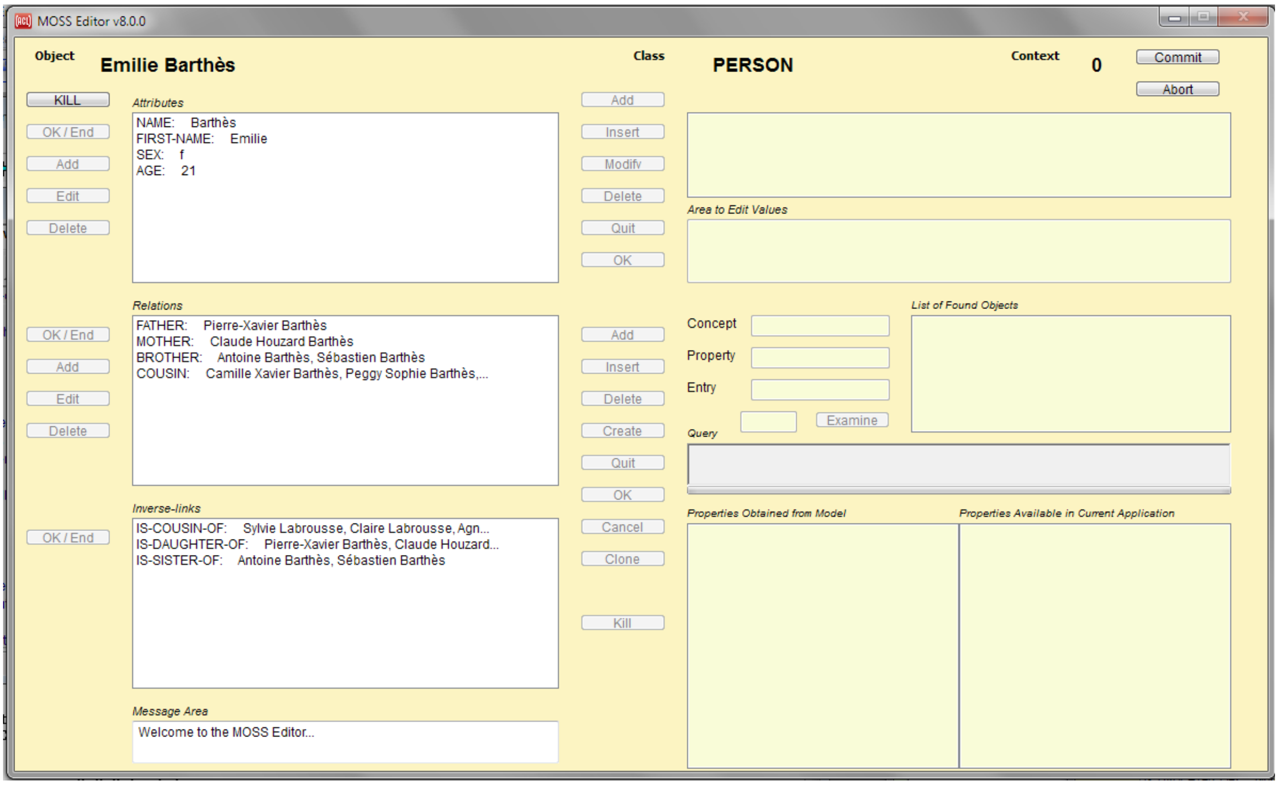

Figure 2.12: MOSS Editor

2.2.4 Other Buttons

We give in this paragraph the meaning of the various buttons of the MOSS window.

Top Row

The top row has four options:

- CONTEXT slot: the slot is used for versioning. It displays the current context of the ontology. One can change the context by typing another value. If the value is legal, then the ontology will be displayed in that context. If the value is illegal, then an error message is displayed and the value is reset to the previous legal one.
- LANGUAGE menu: allows to change the language of the ontology for multilingual ontologies.
- PACKAGE slot: contains the name of the current package. Normally it is the same as the name of the ontology.
- EXIT button: closes the MOSS WINDOW.

Intermediate Buttons

The meaning for the buttons are from left to right:

- OVERVIEW: opens a window for displaying the ontology and knowledge base.
- DB LOAD ALL: unused yet.

- DB STRUCTURE: when a database is opened, shows the different partitions of the database, knowing that a partition corresponds to an ontology.
- DB CONTENT: prints all the keys of the database without the associated values (used for debugging).
- EXECUTE is mostly intended for debugging and executes any Lisp expression typed into the INPUT PANE, displaying the result into the ANSWER PANE.
- CLEAR ANSWER obvious role is to clear the ANSWER PANE.
- ABRT QRY: if a query takes too long to execute (according to the progression of the thermometer) allows interrupting it.
- BROWSE displays a selected object among the objects returned as the result of a query in a BROWSER WINDOW.
- CLEAR ANSWER: clears the answer pane.

Middle Check Boxes

- clean panes: if checked cleans the answer pane after each query (default). Otherwise, prints the answers at the end of the previous ones.
- trace queries: prints details about how the query is processed, in particular shows how initial candidates are filtered out.
- (unused): this check box is reserved for future developments.

Central Mode Radio Buttons

The buttons control the mode of interaction.

- Query: the input pane accepts MOSS queries. When the query is complete, a carriage return executes it. In this mode, Lisp expressions can be executed by pressing the EXECUTE button.
- MOSS help: Set the MOSS WINDOW in an interactive dialog mode. The user can ask MOSS help by typing natural language phrases.
- Data: In this mode the input pane acts as a Lisp listener.

2.3 MOSS Concept Display

The Concept Display as shown Fig.2.13 is used to browse application concepts and individuals. It has three main areas: a left concept area, a central instance area for displaying individuals, and a right detail area.

2.3.1 Concept Display Concept Area

The left part contains a tree representation of the application concepts/subconcepts, actually it is a forest. Top level concepts are represented as files, and concepts with subconcepts as folders.

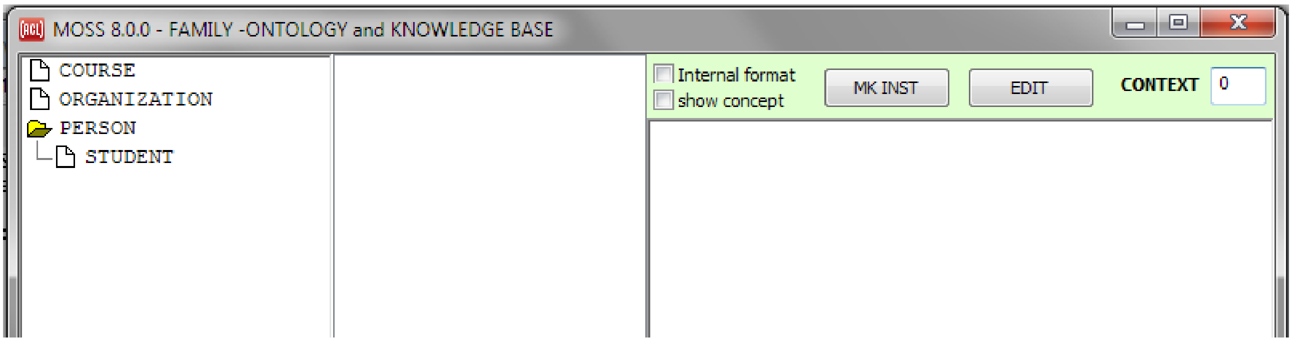

Figure 2.13: MOSS concept display

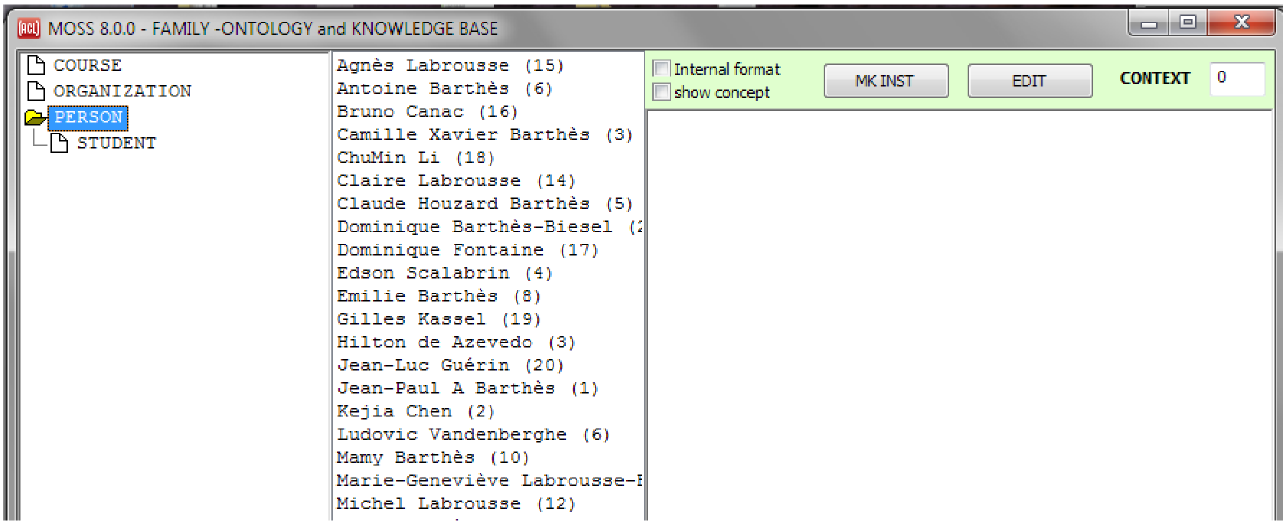

Figure 2.14: The PERSON concept and corresponding individuals

2.3.2 Concept Display Central Area

When one selects a concept, the list of corresponding individuals is displayed in the central area (Fig.2.14).

Individuals corresponding to a concept include individuals corresponding to subconcepts. Individuals are displayed using the =summary method attached to the concept, or else as a list of internal identifiers when the method has not been defined (Fig.2.15).

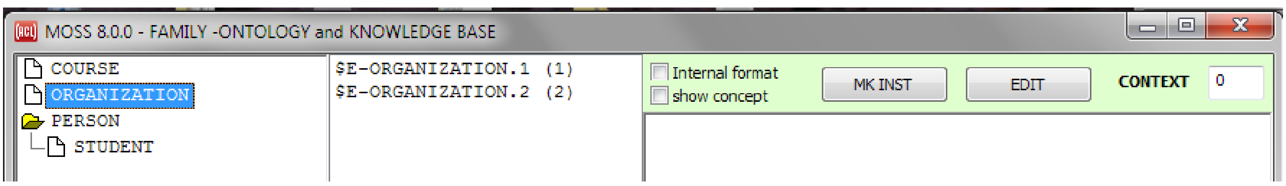

Figure 2.15: The ORGANISM concept and corresponding individuals (internal identifiers)

2.3.3 Concept Display Detail Area

When an individual is selected the details are shown on the right hand side detail area (Fig.2.16).

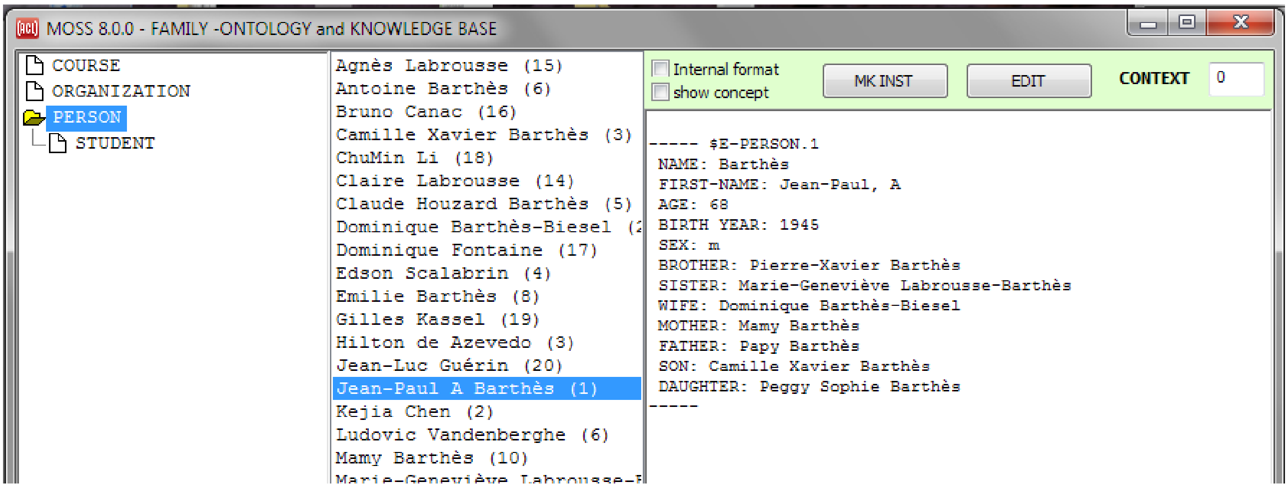

Figure 2.16: Details of the $E-PERSON.1 object

It is possible to view the internal format of an object by checking the INTERNAL FORMAT checkbox. However, this is not intended for the casual user (Fig.2.17).

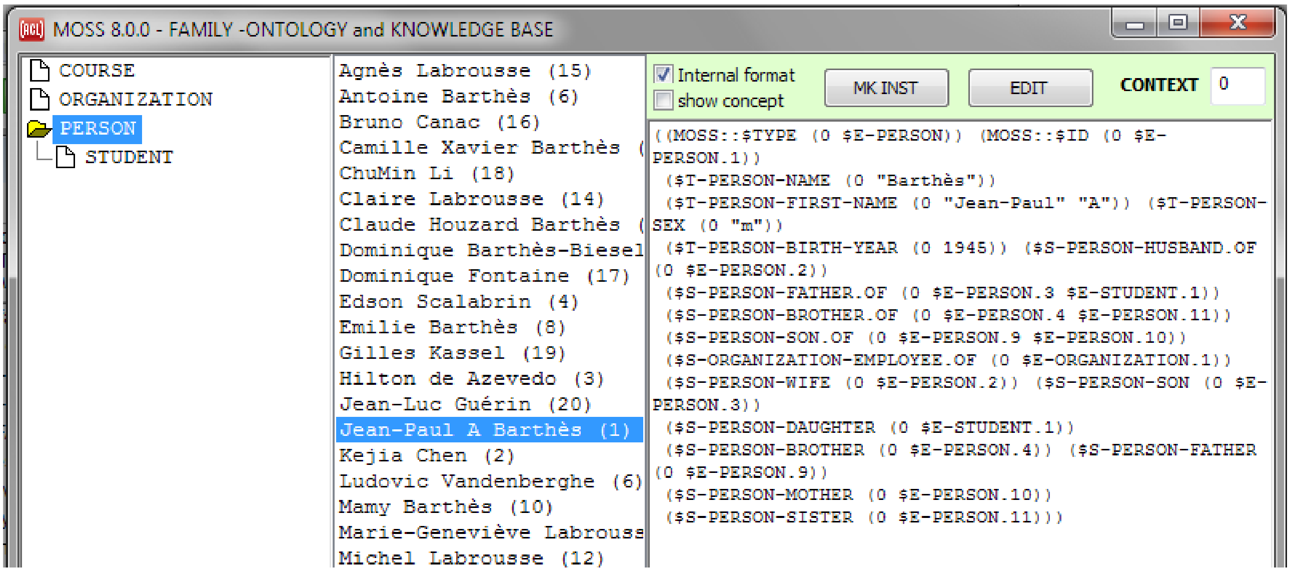

Figure 2.17: Internal representation of the $E-PERSON.1 object

It is also possible to view the concept itself rather than the selected individual by checking the show concept check box (Fig.2.18).

It is also possible by checking both check boxes to view the internal format of the concept PERSON.

2.3.4 Editing or Creating Objects

This can be done by clicking the EDIT button. The EDITOR window is then called.

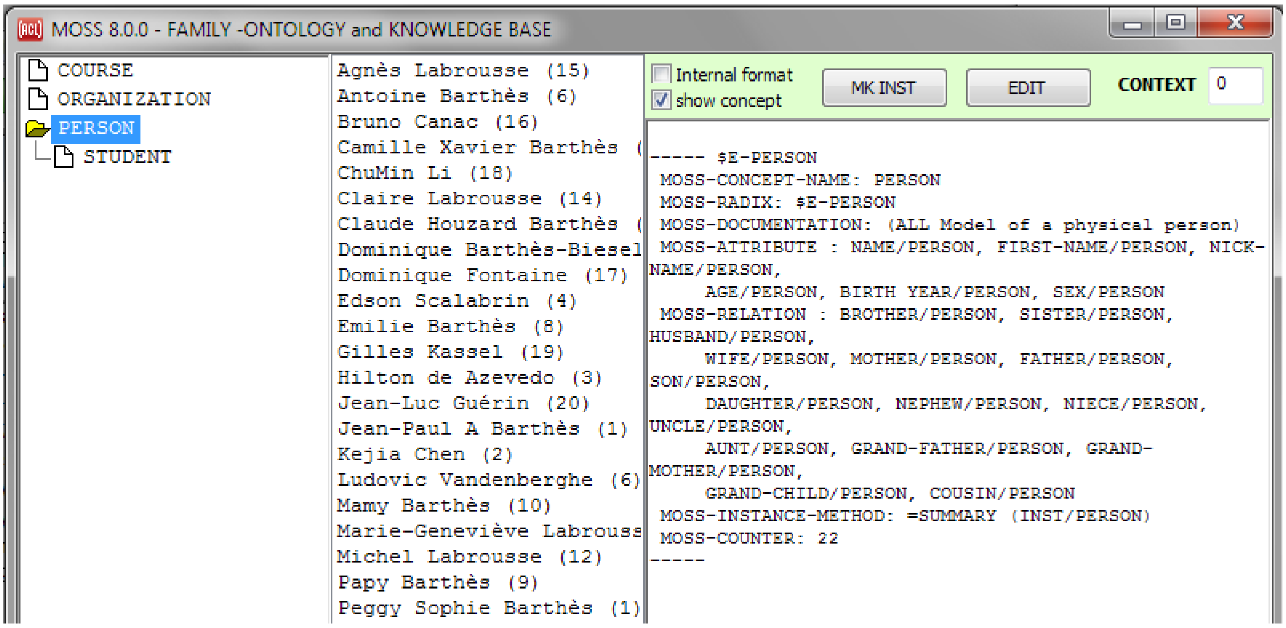

Figure 2.18: Detail of the PERSON concept

It is possible to create a new individual by clicking the "MK INST" button. The EDITOR opens with a new empty individual of the selected concept.

However, in both cases the changes will occur in memory and will not be reported back to the original text file defining the knowledge base.

2.4 The MOSS Browser

The MOSS browser is called when clicking the BROWSE button in the MOSS window (Fig.2.19).

The MOSS browser window displays the title of the object (concept or individual) as the result of applying the =summary method to the object. It contains three areas: one for attributes, one for relations, and one for inverse relations (inverse links). The context in which the object is displayed is shown at the top right corner, and an Edit button is available at the top left corner.

2.4.1 Browsing the Knowledge Base

MOSS allows navigation as follows:

Clicking on a relation or on an inverse link opens a small temporary window containing information about the linked objects (Fig.2.20). The first value is the internal key of the object (here $E-PERSON.9).

Double clicking on one of the objects of the temporary window opens a new window with the details of the new object. Windows are tiled.

Coming back to the previous window is done by closing the browser window.

Thus the user can navigate from object to object using direct or inverse links and stop for editing some of the objects.

2.4.2 Launching the Editor

Opening an Editor for the current browsed object is done by clicking the "Edit" button (Fig.2.21).

Using the browser is very intuitive.

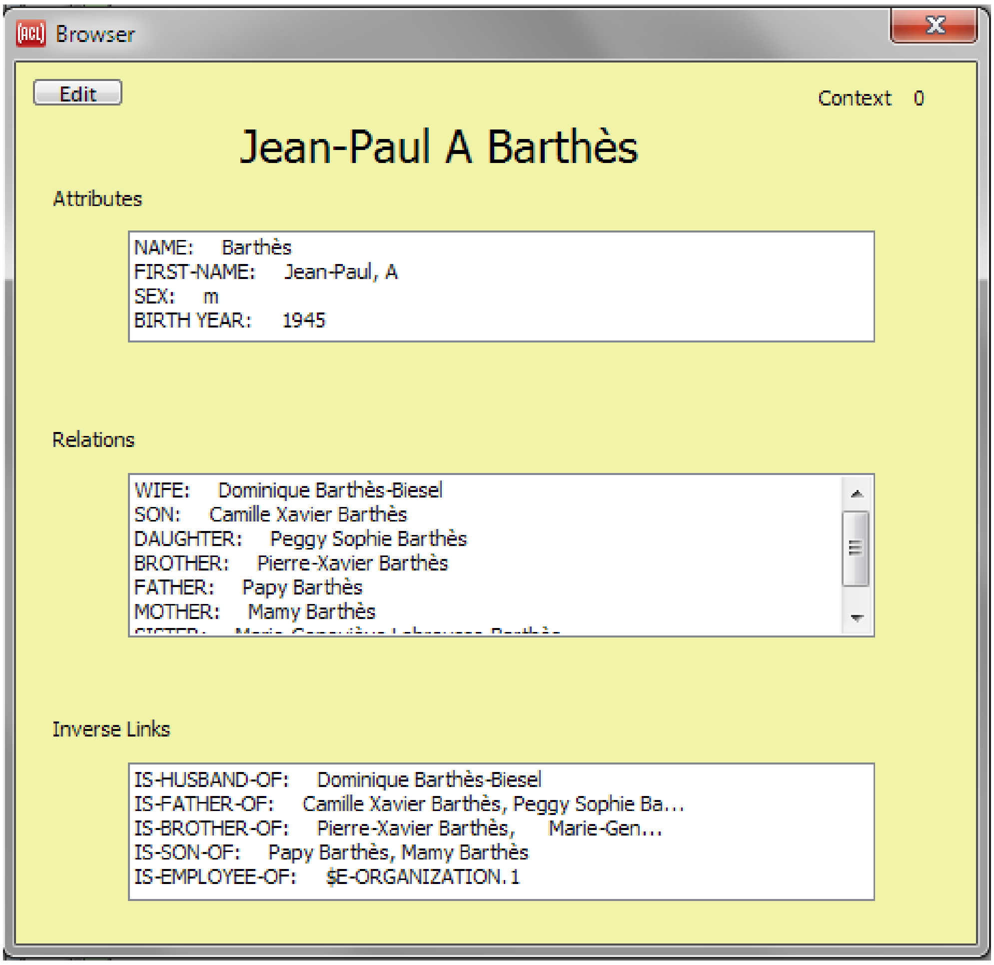

Figure 2.19: The MOSS browser window

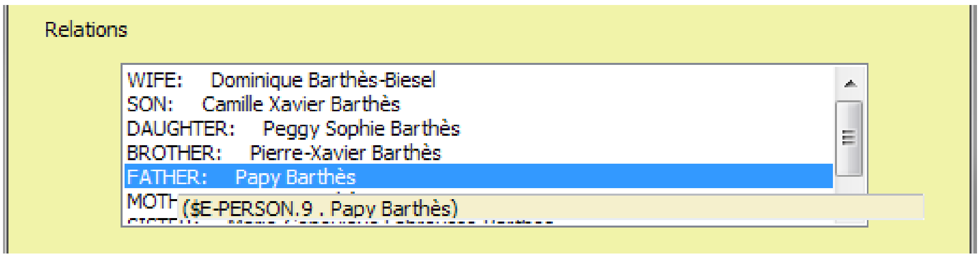

Figure 2.20: Objects linked to the current object

2.5 The HELP Dialog

MOSS contains an experimental version of an online help dialog using natural language. The dialog is started when the help mode is selected. In the help mode one can type natural language requests into the input pane of the MOSS window. Such requests are analyzed by MOSS and an answer is eventually produced. Examples of requests are:

- What can you do for me? (Fig.5.2)
- What is a concept? (Fig.2.23)

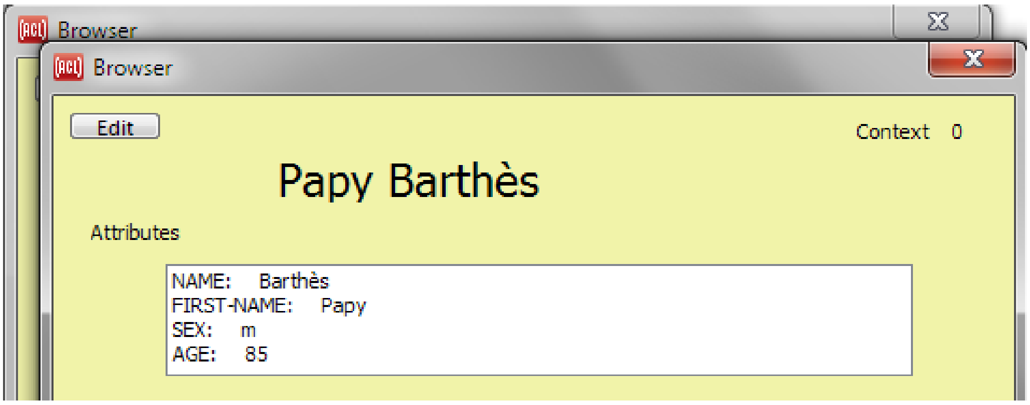

Figure 2.21: Edit and Exit button

- Give me an example of concept. (Fig.2.24)
- Show me a concept.
- How do I create an attribute? (Fig.2.25)

Note that the input sentence must end with a period or a question mark that will not appear on the screen. The syntax needs not be exact and approximate questions, although syntactically incorrect, can be answered, e.g.

- How create attribute?

The sentence contains enough information to be processed.

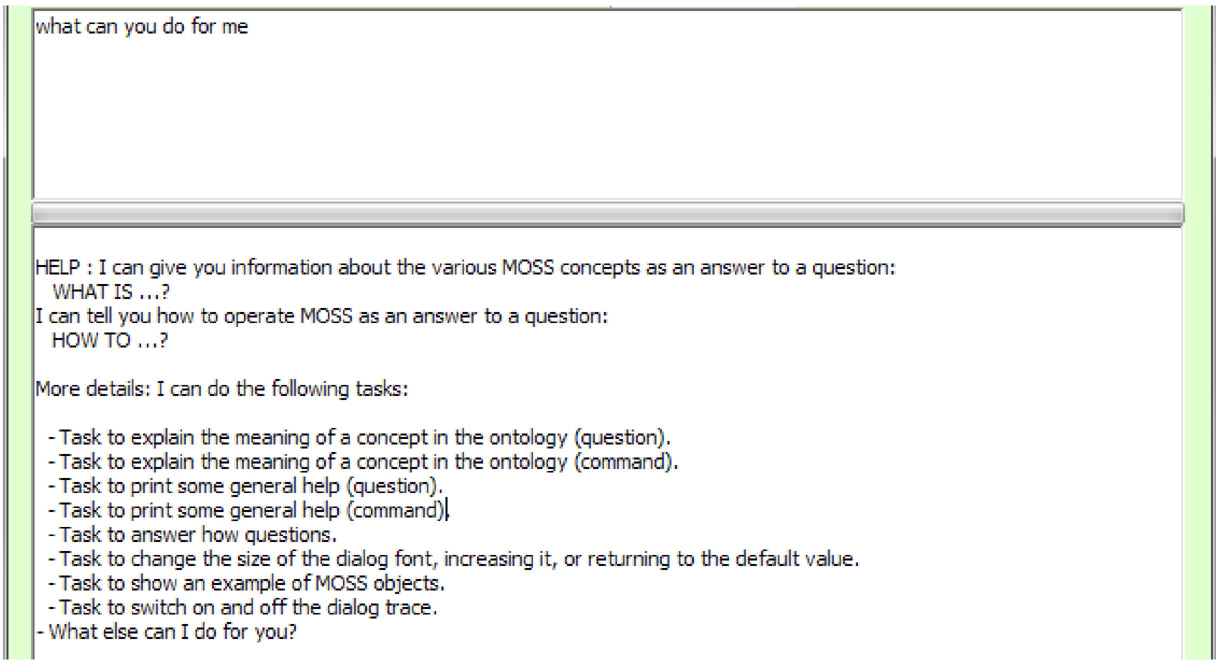

Figure 2.22: General question in the HELP dialog

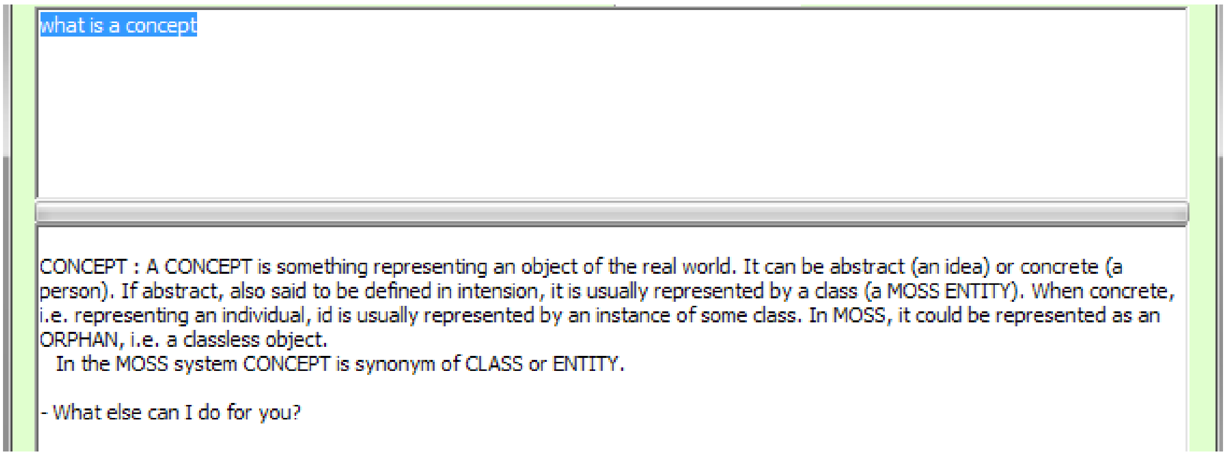

Figure 2.23: Asking for the definition of some MOSS concept

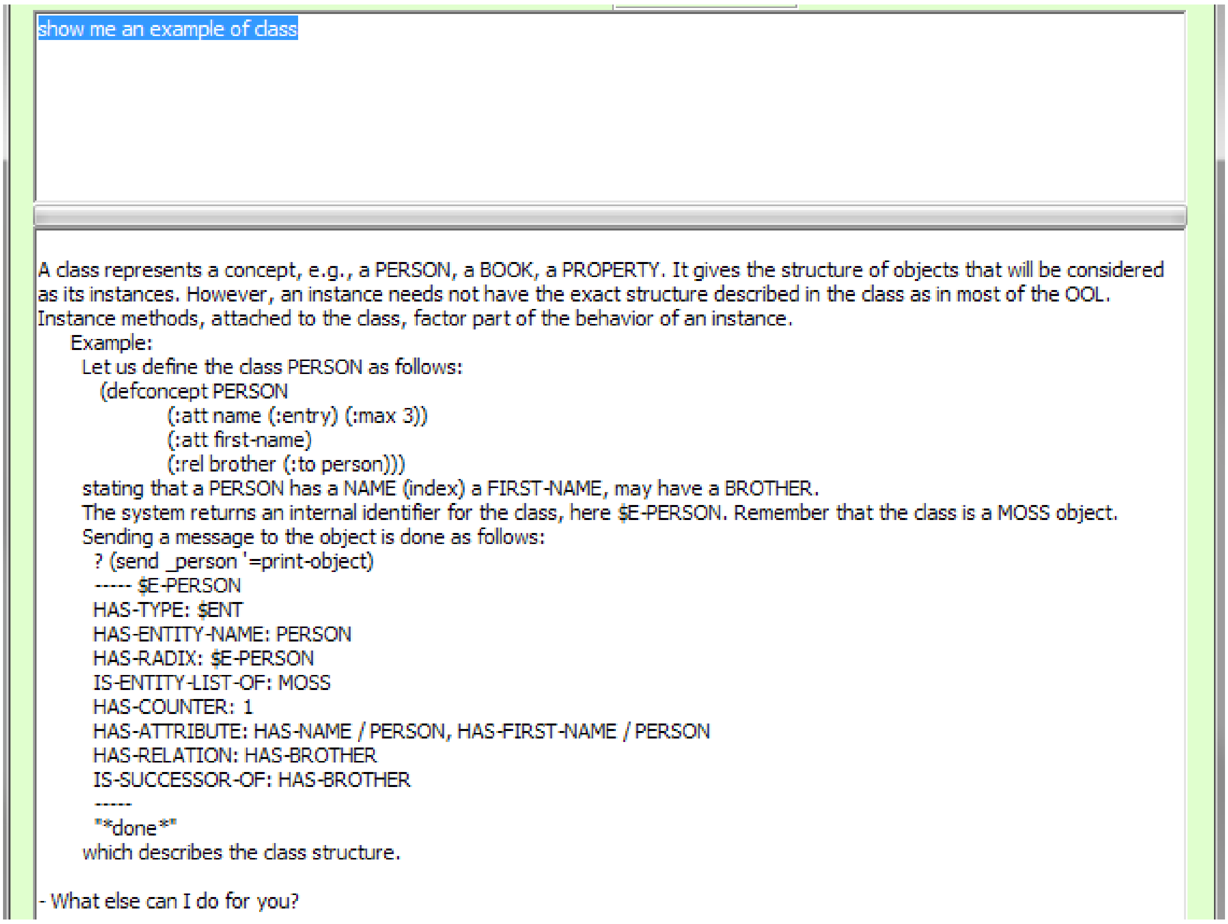

Figure 2.24: Displaying an example of class (concept)

Note The reader must be warned that the HELP system is quite incomplete and needs be upgraded with enough (canned) answers to be of real help.

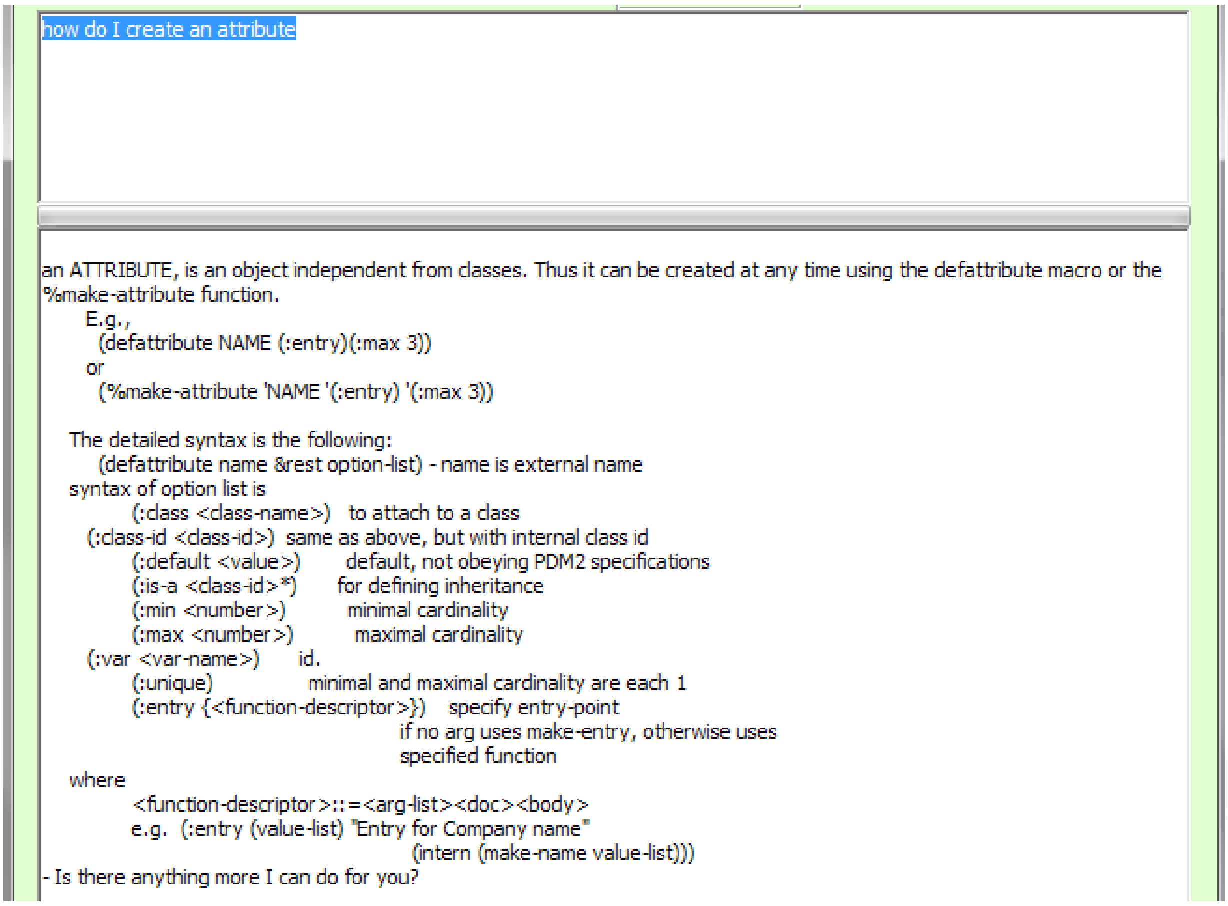

Figure 2.25: How to create an attribute

2.6 MOSS Command Window

Previous sections detailed how to use various windows to explore MOSS ontologies and knowledge bases. The MOSS COMMAND WINDOW (Fig.2.26) is used to open, load, import, save or export the corresponding ontologies and knowledge bases.

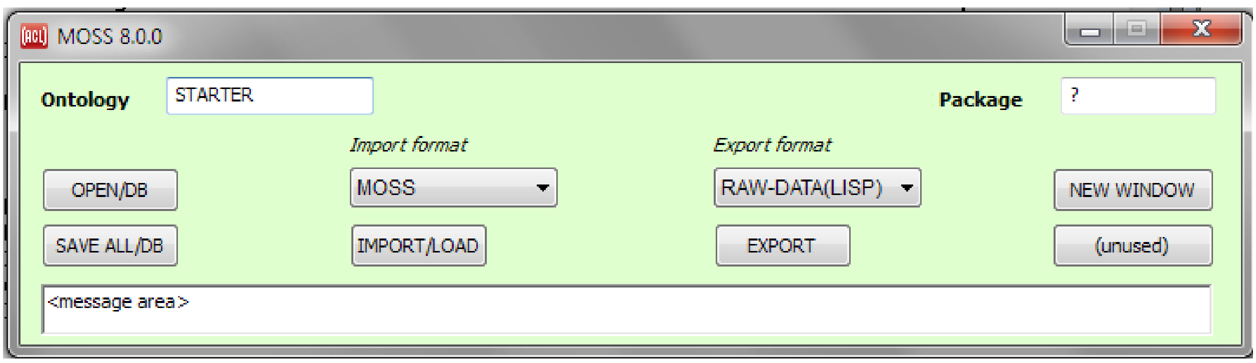

Figure 2.26: MOSS COMMAND window

The meaning of the various areas and buttons are explained as follows.

Top Row

The top row contains two areas:

- Ontology: contains the name of the ontology to be examined or loaded. If the ontology is in a text file, then it can be loaded setting the input format and clicking the IMPORT/LOAD button. If the ontology is in a database, it can be loaded by clicking the OPEN/DB button. When MOSS is starter, this slot shows the default (empty) application called STARTER.
- Package: Name of the ontology package. When MOSS is started the slot contains a question mark. If nothing is done, the ontology package will have the same name than the ontology.

Button Rows

- OPEN/DB: the button is used to open a database containing the ontology and knowledge base. The default database is called MOSS-ONTOLOGIES. Each ontology (concepts and individuals) are stored in a separate partition of the database. When clicking the button a pop up window appears asking if the default database is wanted (Fig.2.27).

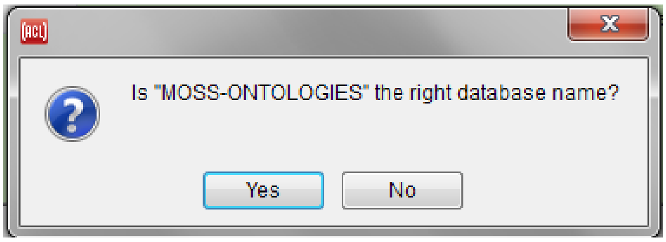

Figure 2.27: Asking for database name

If the wanted database is not the default database, then a selection windows appears in order to locate the proper database (Fig.2.28).

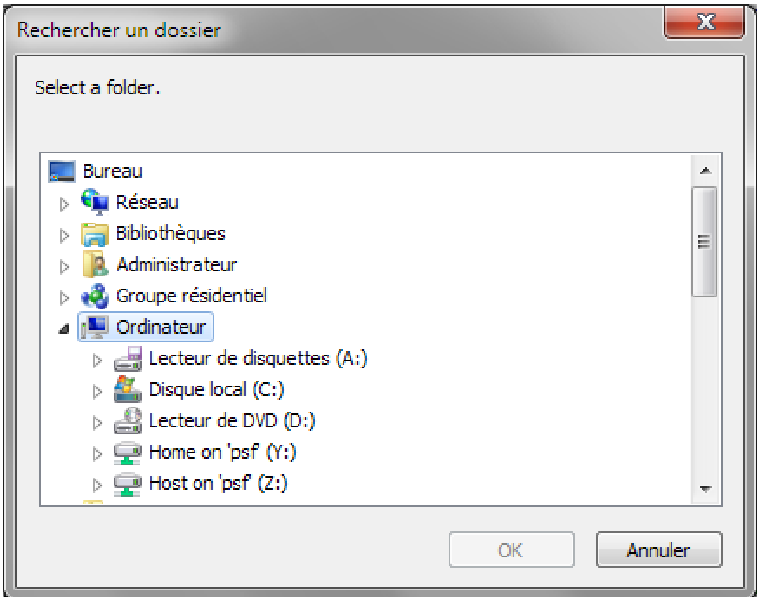

Figure 2.28: Pop up window to locate the ontology database

- SAVE ALL/DB: the button is used to save an ontology that has been loaded from a text file. MOSS asks the user if the target database is the default database, or if another database is intended. If the database does not exist, it will be created. Once an ontology is saved into a database and changes are brought to the ontology, its content becomes different from that of the initial text file.
- Import format: a menu to choose the input format. Currently the only input format is MOSS.
- IMPORT/LOAD: loads the ontology from a text file with the specified input format.
- Export format: export format to save the ontology into a text file. Currently the file is dumped as a series of Lisp expressions.
- NEW WINDOW: clicking the button creates a new MOSS window.

Bottom Area

The bottom area is used to print messages to the user.

2.7 Examples of MOSS Ontologies and Knowledge Bases

The following sections contain two examples of knowledge bases. The first one, FAMILY, deals with family links that are complex enough to exercise the possibilities of querying the knowledge base, the second one, V3S, comes from an OMAS application, and shows how to organize an ontology that can be submitted to the SOL compiler in order to produce an OWL output as well as various files (browser, grapher, ...).

2.7.1 The FAMILY Knowledge Base

The family knowledge base is the one used in this document. It is a fairly old test file that contains persons, organisms and links between such persons and organisms. Family links are complex enough to generate interesting queries.

The family knowledge base is produced in the "FAMILY" package. Four concepts are defined. Two macros, mp and ms, are defined to ease the input of individuals, a small expert system is defined to close the family links.

```
;;;-*- Mode: Lisp; Package: "FAMILY" -*-
;;;===============================================================================
;;;10/02/23
;;; F A M I L Y (File FAMILY-Mac.LSP)
;;;
;;; Copyright Jean-Paul Barthès @ UTC 1994-2010
;;;
;;;===============================================================================

;;; THIS FILE CANNOT BE LOADED FROM THE MOSS-WINDOW. It must be loaded manually.

;;; This file creates a family for checking the LOB window system, and the query
;;; system. Thus it requires the corresponding modules.
;;;1995
;;;  2/13 Adding relationships in the family to be able to use the file for
;;;       testing the query system
;;;       Adding other persons not actual members of the family
;;;  2/18 Adding a mechanism for saturating the family links -> v 1.1.1
;;;  2/19 Adding the age property to all persons and birth date (year)
;;;       Changing the version name to comply with the curreunt MOSS version 3.2
;;;       -> v 3.2.2
;;;  2/25 Adding orphans -> v 3.2.3
;;;1997
;;;  2/18 Adding in-package for compatibility with Allegro CL on SUNs
#|
2003
 0517 used to test changes to the new MOSS v4.2a system
2004
 0628 removing ids from the class definitions
 0715 adding :common-graphics-user-package for ACL environments
 0927 changinging the é and é characters to comply with ACL coding...
2005
 0306 defining a special file for mac because of the problem of coding the French
      letters
```

```
 1217 Changing the file to adapt to Version 6.
2006
 0822 adding count to the inference loop
2007
 0305 cosmetic changes in the documentation
2008
 0520 changing defownmethod syntax
2010
 0209 changing syntax to align with MOSS version 8.0
|#

(defpackage :family (:use :cl :moss))

(moss::mformat "~%;*** loading Family 8.0 Tests ***")

;;; remove tracing while objects are built
;(moss::toff)

;;; trace dialog
(setq moss::*verbose* t)
(setq moss::*dialog-verbose* t)

;(setq moss::*allow-forward-references* nil)

(defontology
  (:title "FAMILY Test Ontology")
  (:package :family)
  (:version "1.0")
  (:language :all)
  )

(in-package :family)

(defconcept (:en "PERSON" :fr "PERSONNE")
  (:doc "Model of a physical person")
  (:tp (:en "NAME" :fr "NOM") (:min 1)(:max 3)(:entry))
  (:tp (:en "FIRST-NAME" :fr "PRENOM"))
  (:tp (:en "NICK-NAME" :fr "SURNOM"))
  (:tp (:en "AGE" :fr "AGE") (:unique))
  (:tp (:en "BIRTH YEAR" :fr "ANNEE DE NAISSANCE") (:unique))
  (:tp (:en "SEX" :fr "SEXE") (:name sex)(:unique))
  (:sp (:en "BROTHER" :fr "FRERE") (:to "PERSON"))
  (:sp (:en "SISTER" :fr "SOEUR") (:to "PERSON"))
  (:rel (:en "HUSBAND" :fr "MARI") (:to "PERSON"))
  (:rel (:en "WIFE" :fr "FEMME") (:to "PERSON"))
  (:rel (:en "MOTHER" :fr "MERE") (:to "PERSON"))
  (:rel (:en "FATHER" :fr "PERE") (:to "PERSON"))
  (:rel (:en "SON" :fr "FILS") (:to "PERSON"))
  (:rel (:en "DAUGHTER" :fr "FILLE") (:to "PERSON"))
  (:rel (:en "NEPHEW" :fr "NEVEU") (:to "PERSON"))
```

```
  (:rel (:en "NIECE" :fr "NIECE") (:to "PERSON"))
  (:rel (:en "UNCLE" :fr "ONCLE") (:to "PERSON"))
  (:rel (:en "AUNT" :fr "TANTE") (:to "PERSON"))
  (:rel (:en "GRAND-FATHER" :fr "GRAND PERE") (:to "PERSON"))
  (:rel (:en "GRAND-MOTHER" :fr "GRAND MERE") (:to "PERSON"))
  (:rel (:en "GRAND-CHILD" :fr "PETITS-ENFANTS") (:to "PERSON"))
  (:rel (:en "COUSIN" :fr "COUSIN") (:to "PERSON"))
  )

(defconcept (:en "COURSE" :fr "COURS")
  (:tp (:en "TITLE" :fr "TITRE") (:unique))
  (:tp (:en "LABEL" :fr "CODE") (:entry))
  (:doc (:en "Course followed usually by students"))
  )

(defconcept (:en "STUDENT" :fr "ETUDIANT")
  (:doc (:en "A STUDENT is a person usually enrolled in higher education"))
  (:is-a "PERSON")
  (:rel (:en "COURSES" :fr "COURS") (:to "COURSE"))
  )

(defconcept (:en "ORGANIZATION" :fr "ORGANISME")
  (:doc (:en "an ORGANIZATION is a body of persons organized for a specific purpose ~
             as a club, union or society."))
  (:tp (:en "NAME" :fr "NOM"))
  (:tp (:en "ABBREVIATION" :fr "SIGLE") (:entry))
  (:rel (:en "EMPLOYEE" :fr "EMPLOYE") (:to "PERSON"))
  (:rel (:en "STUDENT" :fr "ETUDIANT") (:to "STUDENT"))
  )



#|
(defownmethod =make-entry ("NAME" HAS-PROPERTY-NAME TERMINAL-PROPERTY
                                   :class-ref PERSON)
  (data &key package)
  "Builds an entry point for person using standard make-entry primitive"
  (declare (ignore package))
  (moss::%make-entry-symbols data))
|#

(defownmethod =if-needed ("AGE" HAS-MOSS-PROPERTY-NAME MOSS-ATTRIBUTE :class-ref PERSON)
  (obj-id)
  (let ((birth-year (car (send obj-id '=get 'HAS-BIRTH-YEAR))))
    (if (numberp birth-year)
      ;; do not forget, we must return a list of values
      (list (- (get-current-year) birth-year)))))

(definstmethod =summary PERSON ()
  "Extract first names and names from a person object"
```

```
  (let ((result (format nil "~{~A~^ ~} ~{~A~^-~}"
                      (g==> "first-name") (g==> "name"))))
    (if (string-equal result " ")
      (setq result "<no data available>"))
    (list result)))

;(format nil "~{~A~^ ~}, ~{~A ~}" '("Barthès" "Biesel") '("Dominique" "Sophie"))

;;; First define a macro allowing to create persons easily

(defMacro mp (name first-name sex var &rest properties)
  `(progn
     (defVar ,var)
     (defindividual "PERSON"
       ("NAME" ,@name)
       ("FIRST-NAME" ,@first-name)
       ("SEX" ,sex)
       (:var ,var)
       ,@properties)))

(defMacro ms (name first-name sex var &rest properties)
  `(progn
     (defVar ,var)
     (defindividual STUDENT
       ("NAME" ,@name)
       ("FIRST-NAME" ,@first-name)
       ("SEX" ,sex)
       (:var ,var)
       ,@properties)))

;;; Instances
(mp ("Barthès") ("Jean-Paul" "A") "m" _jpb ("BIRTH YEAR" 1945))
(mp ("Barthès" "Biesel") ("Dominique") "f" _dbb ("HUSBAND" _jpb)("BIRTH YEAR" 1946))
;(send _dbb '=print-self)
;(break "family")
(mp ("Barthès") ("Camille" "Xavier") "m" _cxb ("FATHER" _jpb) ("MOTHER" _dbb)
    ("BIRTH YEAR" 1981))
(ms ("Barthès") ("Peggy" "Sophie") "f" _psb ("FATHER" _jpb) ("MOTHER" _dbb)
    ("BROTHER" _cxb)("BIRTH YEAR" 1973))
(mp ("Barthès") ("Pierre-Xavier") "m" _pxb ("BROTHER" _jpb)("AGE" 54))
(mp ("Barthès") ("Claude" "Houzard") "f" _chb ("HUSBAND" _pxb)("AGE" 52))
(mp ("Barthès") ("Antoine") "m" _ab ("FATHER" _pxb)("MOTHER" _chb)
    ("COUSIN" _cxb _psb)("AGE" 27))
(mp ("Barthès") ("Sébastien") "m" _sb ("FATHER" _pxb)("MOTHER" _chb)
    ("BROTHER" _ab)("COUSIN" _cxb _psb)("AGE" 24))
(mp ("Barthès") ("Emilie") "f" _eb ("FATHER" _pxb)("MOTHER" _chb)
    ("BROTHER" _ab _sb)("COUSIN" _cxb _psb)("AGE" 21))
(mp ("Barthès") ("Papy") "m" _apb ("SON" _jpb _pxb)("AGE" 85))
(mp ("Barthès") ("Mamy") "f" _mlb ("SON" _jpb _pxb)("HUSBAND" _apb)("AGE" 81))
(mp ("Labrousse" "Barthès") ("Marie-Geneviève") "f" _mgl ("FATHER" _apb)
```

```
    ("MOTHER" _mlb)("AGE" 48)("BROTHER" _jpb _pxb))
(mp ("Labrousse") ("Michel") "m" _ml ("WIFE" _mgl)("AGE" 52))
(mp ("Labrousse") ("Sylvie") "f" _sl ("FATHER" _ml)("MOTHER" _mgl)
    ("COUSIN" _psb _cxb _ab _sb _eb)("AGE" 19))
(mp ("Labrousse") ("Claire") "f" _cl ("FATHER" _ml)("MOTHER" _mgl)("SISTER" _sl)
    ("COUSIN" _psb _cxb _ab _sb _eb)("AGE" 18))
(mp ("Labrousse") ("Agnès") "f" _al ("FATHER" _ml)("MOTHER" _mgl)
    ("SISTER" _sl _cl)("COUSIN" _psb _cxb _ab _sb _eb)("AGE" 15))
(mp ("Canac") ("Bruno") "m" _bc)

;---
(ms ("Chen") ("Kejia") "f" _kjc)
(ms ("de Azevedo") ("Hilton") "m" _hda)
(ms ("Scalabrin") ("Edson") "m" _es)
(ms ("Marchand") ("Yannick") "m" _ym)
(ms ("Vandenberghe") ("Ludovic") "m" _lv)
;---
(mp ("Fontaine") ("Dominique") "m" _df)
(mp ("Li") ("ChuMin") "m" _cml)
(mp ("Kassel") ("Gilles") "m" _gk)
(mp ("Guérin") ("Jean-Luc") "m" _jlg)
(mp ("Trigano") ("Philippe") "m" _pt)

(defindividual ORGANIZATION ("ABBREVIATION" "UTC")
  ("NAME" "Université de Technologie de Compiègne")
  ("EMPLOYEE" _jpb _dbb _df _gk _pt)
  ("STUDENT" _kjc _hda))

(defindividual ORGANIZATION  ("ABBREVIATION" "IC")
  ("NAME" "Imperial College")
  ("STUDENT" _psb))

(moss::mformat "~%*** saturating family-links ***")

;(setq *persons* (send *qh* '=process-query '(person)))
;;; cannot execute this before instances are initialized since variables like _jpb
;;; are not yet bound
(push
   '(defParameter *persons*
        (list _jpb _dbb _cxb _psb _pxb _chb _ab _sb _eb _apb _mlb _mgl _ml _sl _cl
               _al _kjc _hda _es _ym _lv _df _cml _gk _jlg _pt))
   moss::*deferred-defaults*)
(defVar *rule-set* nil)
(setq *rule-set* nil)

(defMacro ar (&rest rule)
  `(push ',rule *rule-set*))

;;;  Rules to close the family links
;;;
```

```
;;; If person P1 has-sex M
;;;              has-brother P2
;;;    then person P2 has-brother P1
(ar "m" has-brother has-brother)
;;;
;;; If person P1 has-sex M
;;;              has-sister P2
;;;    then person P2 has-brother P1
(ar "m" has-sister has-brother)
;;;
;;; If person P1 has-sex M
;;;              has-father P2
;;;    then person P2 has-son P1
(ar "m" has-father has-son)
;;;
;;; If person P1 has-sex M
;;;              has-mother P2
;;;    then person P2 has-son P1
(ar "m" has-mother has-son)
;;;
;;; If person P1 has-cousin P2
;;;    then person P2 has-cousin P1
(ar "m" has-cousin has-cousin)
(ar "f" has-cousin has-cousin)
;;;
;;; If person P1 has-sex M
;;;              has-son P2
;;;    then person P2 has-father P1
(ar "m" has-son has-father)
;;;
;;; If person P1 has-sex M
;;;              has-daughter P2
;;;    then person P2 has-father P1
(ar "m" has-daughter has-father)
;;;
;;; If person P1 has-sex M
;;;              has-wife P2
;;;    then person P2 has-husband P1
(ar "m" has-wife has-husband)
;;;
;;; If person P1 has-sex F
;;;              has-brother P2
;;;    then person P2 has-sister P1
(ar "f" has-brother has-sister)
;;;
;;; If person P1 has-sex F
;;;              has-sister P2
;;;    then person P2 has-sister P1
(ar "f" has-sister has-sister)
;;;
```

```
;;; If person P1 has-sex F
;;;              has-father P2
;;;    then person P2 has-daughter P1
(ar "f" has-father has-daughter)
;;;
;;; If person P1 has-sex F
;;;              has-mother P2
;;;    then person P2 has-daughter P1
(ar "f" has-mother has-daughter)
;;;
;;; If person P1 has-sex F
;;;              has-son P2
;;;    then person P2 has-mother P1
(ar "f" has-son has-mother)
;;;
;;; If person P1 has-sex F
;;;              has-daughter P2
;;;    then person P2 has-mother P1
(ar "f" has-daughter has-mother)
;;;
;;; If person P1 has-sex F
;;;              has-husband P2
;;;    then person P2 has-wife P1
(ar "f" has-husband has-wife)


;;; the version using the MOSS service functions does not seem significantly
;;; faster than the one using the messages

(defUn apply-rule (p1 p2 rule)
  "produces a list of messages to send to modify the data.
Arguments:
   p1: person 1
   p2: person 2
   rule: (sex) rule to be applied
Return:
   nil or message to be sent."
  ;; note that we use =get and not =get-id that would require the local id of
  ;; the specified property, e.g. $t-person-name or $t-student-name, which
  ;; cannot be guarantied in a global rule
  (let ((sex (car rule))
        (sp-name (cadr rule))
        (ret-sp-name (caddr rule)))
    ;; if P1's sex is that of rule
    ;;    and P1 is linked to P2 by property sp-name
    ;; then
    ;;    link P2 to P1 by property ret-sp-name
    ;; e.g.
    ;;   "f" has-daughter has-mother
    ;;   if P1 is the daughter of P2, add that P2 is the mother of P1
```

```
    (if (and (equal sex
                    (car (moss::%get-value
                          P1
                          (moss::%get-property-id-from-name P1 'has-sex))))
             (member P2 (moss::%get-value
                         P1
                         (moss::%get-property-id-from-name P1 sp-name))))
      `(send ',P2 '=add-related-objects ',ret-sp-name ',P1))))

#|
(apply-rule _pxb _jpb '("m" HAS-BROTHER HAS-BROTHER))
|#

(defUn close-family-links (family-list rule-set)
  "Function that takes a list of persons and computes all the links that must be
added to such persons to close the parent links brother, sister, father, mother,
son, daughter, cousin. It executes a loop until no more changes can occur, applying
rules from a rule set. The result is a list of instructions to be executed."
  (let ((change-flag t) action action-list (count 0))
    (print count)
    ;; Loop until no more rule applies
    (loop
      (unless change-flag (return nil))
      ;; reset flag
      (setq change-flag nil)
      ;; loop on couple of objects in the family-list
      (dolist (P1 family-list)
        (dolist (P2 family-list)
          ;; loop on each rule
          (dolist (rule rule-set)
            ;; if the rule applies it produces an instruction to be executed
            (setq action (apply-rule P1 P2 rule))
            ;; which is appended to the instruction list
            (when (and action
                       (not (member action  action-list :test #'equal)))
              ;; print a mark into the debug trace to tell user we are doing
              ;; some work
              (moss::mformat "*")
              (incf count)
              (if (eql (mod count 50) 0) (moss::mformat "~%~S" count)) ; new line after 50 *s
              (push action action-list)
              ;; in addition we mark that a changed occured by setting the change flag
              (setq change-flag t))))))
    ;; return list of actions
    (reverse action-list)))

(moss::mformat "~%*** closing family links (takes some time) ***")

;;; When loading the file with m-load, a lot of things are deferred and executed
;;; once the file has been loaded, in particlar instance links.
```

```
;;; Thus, the saturation of the instance links cannot be done during the loading of
;;; the file but must also be deferred.

(push '(mapcar #'eval (close-family-links *persons* *rule-set*))
       moss::*deferred-defaults*)

#|
(defVar instructions)

(eval-when (load compile execute)
  (moss::mformat "~%; family-file / persons:~%~S" *persons*)
  (moss::mformat "~%; family-file / rule set:~%~S" *rule-set*)
  (setq instructions (close-family-links *persons* *rule-set*))
  (format t "~%; family-file / instructions for closing links:~%~S" instructions)
  (mapcar #'eval instructions)
  )

(sleep 5)
|#
;;; Adding orphans

(defobject ("NAME" "Dupond")("sex" "m"))
(defobject ("NAME" "Durand" )("SEX" "f")("AGE" 33))

#|
(make-lob-window)
|#

(moss::mformat "~%*** family file loaded ***")
```

2.7.2 The V3S Ontology and Knowledge Base

The V3S knowledge base is a small example of a knowledge base used by an OMAS agent named COLOMBO in charge of modeling an industrial environment. The small extract is intended to indicate some features of the ontology file.

The following points should be noticed:

- The ontology is defined in the name space (package) :colombo.
- The ontology uses the :moss and :omas name spaces.
- A `defontology` macro gives the name of the ontology and the language in which it is defined (here French):

```
(defontology
  (:title "COLOMBO ontology")
  (:version "1.0")
  (:language :fr)
  )
```

- The ontology uses `defchapter` and `defsection` macros to organize the ontology as a knowledge book.

- Actions contain default rules that are interpreted by a special inference engine contained in the COLOMBO agent.

```
;;;-*- Mode: Lisp; Package: "COLOMBO" -*-
;;;===============================================================================
;;;08/02/03
;;;             AGENT COLOMBO: ONTOLOGY (file COLOMBO-ontology.lisp)
;;;             Copyright Barthes@UTC, 2008
;;;
;;;===============================================================================

;;; the idea is to model objects of the environment in a separate agent: COLOMBO
;;; Each agent modelling a specific human, will send order or inquiries to COLOMBO
;;; COLOMBO maintains the model representing the environment, regardless of what
;;; each agent may believe
;;; Actions are represented by methods that hat preconditions and consequences
;;; when the application is successful or if it is a failure
;;; Preconditions are tested typically by checking the status (vale) of some
;;; property of some object. This can be done by applying a query to the object base

(eval-when (:compile-toplevel :load-toplevel :execute)
  (unless (find-package :COLOMBO) (make-package "COLOMBO")))

(in-package :COLOMBO)

(eval-when (compile load eval)
  (use-package :moss)
  (use-package :omas)
  ) ; reexport it for use in other packages

(defontology
  (:title "COLOMBO ontology")
  (:version "1.0")
  (:language :fr)
  )

;;;======================================================================= ACTIONS

(defchapter
  (:name :en "mv-actions"
         :fr "actions MV"))

(defconcept "mv-action"
  (:att "nom" (:entry) (:unique))
  (:att "position")
  (:att "orientation")
  (:att "rules")
  (:att "method" (:default =close-gate))
  )
```

```
;;;========================================================================== OBJETS

(defchapter
  (:name :en "objects"
         :fr "objets"))

;;;---------------------------------------------------- OBJETS OUVRABLES

(defsection
  (:name  :fr "objet ouvrable"))

(defconcept "objet-ouvrable"
  )

;;;------------------------------------------------------------------------- VANNE

(defconcept "vanne"
  (:is-a "objet-ouvrable")
  (:att "identifiant" (:unique)(:entry))
  (:att "position" (:unique))
  (:att "orientation" (:unique) (:default '(1 0 0 0)))
  ;(:rel "mv-action" (:to "mv-action"))
  (:att "etat" (:one-of "normal" "grippe" "casse")(:default "normal"))
  (:att "statut" (:one-of "ouvert" "ferme" "inconnu")(:default "inconnu"))
  (:doc :fr "une VANNE est un dispositif present sur une canalisation qui controle ~
            le flux de liquide dans celle-ci.")
  )

(definstmethod =summary VANNE ()
  (HAS-IDENTIFIANT))

;;;---------------------------------------------------------------------- ACTIONS

(defchapter
  (:name :en "mv-actions"
         :fr "actions MV"))

;;;---------------------------------------------------------- OUVRIR VANNE

(defindividual "mv-action"
  ("nom" "ouvrir vanne")
  ("rules"
   (:if (?* "vanne" ("statut" :is "ouvert"))
     :then
     (:return :success))
   (:if (?* "vanne" ("statut" :is "ferme"))
     (?* "vanne" ("etat" :is-not "grippe"))
     :then
     (:set ?* "statut" "ouvert")
     (:return :success))
```

```
   (:if
     (?* "vanne" ("statut" :is "grippe"))
     :then
     (:return :failure))
   )
  (:var _ouvrir-vanne))

(definstmethod =summary MV-ACTION ()
  (HAS-NOM))

;;;------------------------------------------------------------ FERMER VANNE

(defindividual "mv-action"
  ("nom" "fermer vanne")
  ("rules"
   (:if (?* "vanne" ("statut" :is "ferme"))
     :then
     (:return :success))
   (:if (?* "vanne" ("statut" :is "ouvert"))
     (?* "vanne" ("etat" :is-not "grippe"))
     :then
     (:set ?* "statut" "ferme")
     (:return :success))
   (:if
     (?* "vanne" ("statut" :is "grippe"))
     :then
     (:return :failure))
   )
  (:var _fermer-vanne))

;;; adding default actions to vanne

(defrelation "mv-action" "vanne" "mv-action"
  (:default _ouvrir-vanne _fermer-vanne))

;;;==========================================================================
;;;
;;;                                INDIVIDUALS
;;;
;;;==========================================================================

(defchapter
  (:name :en "OBJECTS"
         :fr "OBJETS"))

;;;------------------------------------------------------------------------ VANNES

(defsection
  (:name :en "gates"
         :fr "vannes"))
```

```
(catch :error (defindividual "vanne"
  ("identifiant" "V234")
  ("statut" "ferme")
  ("etat" "grippe")
  (:var _vanne234)))

(defindividual "vanne"
  ("identifiant" "V345")
  ("statut" "ouvert")
  (:var _vanne345))

(defindividual "vanne"
  ("identifiant" "V101")
  ("statut" "ferme")
  (:var _vanne101))

:EOF
```

Chapter 3

MOSS Syntax

Contents

3.1 Introduction

3.1.1 For Beginners

This manual describes how to use the MOSS syntax, e.g. for designing ontologies. Beginners should also refer to Chapter 1.

3.1.2 Global Information

User space is `:common-graphics-user` in ACL (Windows). Creating objects is done by means of LISP macros. Errors during execution can be caught by wrapping the macro execution with a

```
(catch :error <macro execution>)
```

In case of error the result of the `catch` is a string that can be printed.

Examples are given with a "?" prompt, replacing the standard "CG-USER(nn):" prompt.

3.1.3 Names, Multilingual Names, References, Internal Names, Identifiers

Starting with version 7, MOSS includes multilingual features, which allows to build ontologies combining several languages. However, one can use a single language, which simplifies definitions. Starting with version 8, we allow several ontologies to be defined in different packages, which makes things more complex. However, we tried to retain a simple way of defining concepts, properties, individual, or other MOSS objects.

Previous versions made intensive use of symbols for naming concepts, properties or other objects. Although we kept this possibility, we tend now to favor strings when using a single language. When using several languages one must use a multilingual name. The concept is introduced in this section.

Thus, MOSS allows different types of names, strings, references.

Strings

Names are the favored way of expressing names, e.g.

```
(defconcept "Government Agency" (:att "Acronym"))
```

After execution the name of the class will be transformed into the following multilingual name: (:EN "Government Agency"), that of the attribute into (:EN "Acronym").

Note: we use here the default value for the language, namely :EN for English.

Multilingual name

A multilingual name is used for defining concepts using different languages, e.g.

```
? (defconcept (:name :en "Person" :fr "Personne")
    (:att (:name :en "Age" :fr "Âge")))
```

Here we used English and French as specified by the tags (`:en` and `:fr`). Note that `:en` is used as a default by the system. Note also that French special letters may require a special encoding like UNICODE UTF-8 (recommended).

Note finally that the `:name` tag in the list is optional. One could write:

```
? (defconcept (:en "Person" :fr "Personne")
    (:att (:en "Age" :fr "Âge")))
```

Name

Names are kept for backward compatibility. However, we strongly encourage to use string references, rather than symbols.

A name corresponds to a Lisp symbol, e.g. `PERSON, AGE`. When used in the defining macros it does not need to be quoted, e.g.

```
(defconcept PERSON (:att AGE))
```

Case is not important, thus the following line has the same effect as the previous one:

```
(defCONCEPT Person (att:  aGE))
```

After execution, the name of the concept will be transformed into the following multilingual name: `(:EN "PERSON")`, that of the attribute into `(:EN "AGE")`[1].
A name will print in capital letters in the various outputs.

References

A reference stands for a name, string or multilingual name. References can be used in many places. When a reference is required, one may use any form for it, e.g.

```
? (defconcept (:en "Person" :fr "Personne")
    (:att "Age")))
```

[1]:EN denoting English is the default language, It is the value of the global `moss::*language*` variable

Synonyms

It is possible to declare any number of synonyms in a string by using a semi column, e.g.

```
?  (defattribute "name ; given name" (:class "student")))
; Warning: reusing previously defined generic attribute HAS-NAME, ignoring eventual options.
; While executing: MOSS::%MAKE-GENERIC-TP
$T-STUDENT-NAME

? has-name
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 HAS-NAME))
 (MOSS::$PNAM.OF (0 $T-NAME $T-STUDENT-NAME)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
? has-given-name
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 HAS-GIVEN-NAME))
 (MOSS::$PNAM.OF (0 $T-STUDENT-NAME)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

Note that we have created 2 internal names for the attribute: `HAS-NAME` and `HAS-GIVEN-NAME`.

Internal Names

MOSS builds internal names for the different objects, creating new symbols.

Concept names repeat the external name as a symbol. Thus, `PERSON` will be the internal name of the PERSON concept, however it was defined. If a string is given containing several words the symbol is created by using hyphenation, e.g. `"Government Agency"` produces `GOVERNMENT-AGENCY`.

Property names are built by prefixing the symbol name with `"HAS-"`. E.g., `"Age"` will produce `HAS-AGE`. Multiple word names are hyphenated, e.g. `HAS-FIRST-NAME`. Inverse property internal names are built using prefix and suffix, e.g. `IS-BROTHER-OF`. Note that prefixes and suffixes are defined using English, which can lead to strange names when using a different language, e.g. `HAS-NOM` or `IS-NOM-OF`! However, internal names are not supposed to be seen by the user and their role is to provide unique identifiers. Thus, mixing languages at this level is not a problem.

Variables

Variables names can be used in the `:var` option for storing identifiers temporarily. They should start with an underscore, e.g.

```
? (defindividual "Person" ("name" "Dupond" "Smith")(:var _ds))
$E-PERSON.2
? _ds
$E-PERSON.2
```

Identifiers

Identifiers are given by the MOSS system automatically and may change depending on the order in which definition macros are executed. Thus, one should not rely on them in application programs. Examples of identifiers:

```
$E-PERSON, $E-STUDENT, $T-PERSON-NAME, $S-STUDENT-BROTHER, $E-PERSON.2,
$S-BROTHER.OF, etc.
```

Macros Options

Macro options are specified as lists starting with a keyword, e.g.

`(:max 3), (:entry), (:one-of "young" "old" "ancient"), etc.`

3.1.4 Namespaces

Namespaces use the concept of Lisp package. All MOSS system objects are defined in the "MOSS" package. By default applications are run in the "COMMON-GRAPHICS-USER" package. However, applications can use their own package (name space). The OMAS multi-agent platform uses a specific package for each agent. Thus each agent can develop its own ontology without interfering with the other agents. For simplified programming one can inherit symbols from the "MOSS" package by inserting a command:

```
(use-package :moss)
```

at the beginning of the application code.

In a programming environment the current namespace is given by the global variable `*package*`, the last defined application package by `moss::*application-package*`.

Note that in a Lisp environment keywords (e.g., `:ref`) and strings do not depend on packages, but symbols do. When defining a new ontology the desired package should be specified by inserting a command:

```
(in-package :my-package)
```

at the beginning of the file, or better a `defontology` macro command. See the documentation about SOL for a detailed treatment of ontologies.

3.1.5 Contexts or Versions

`All MOSS objects` can be versioned. The mechanism is implemented through a configuration graph that records the links between different versions called contexts in the MOSS jargon.

Versioning can be sequential or parallel. Sequential versioning allows accounting for changes in time easily, parallel versioning can be used for trying various hypothesis after an initial situation.

Versioning introduces a new level of complexity in the programming of functions. In a given programming environment the current version is recorded by a number associated to the global variable `*context*`

Note The initial context, root of the configuration tree, is context 0. If no versioning is required then all objects belong to context 0.

The following sections give examples of how to use the defining macros to create MOSS objects. Examples are taken from the ACL environment on the Windows 7 system in the "COMMON-GRAPHICS-USER" package (also known as `:cg-user`).

The syntax and corresponding examples will be given using English as a default language. More complex multilingual examples will be given in Section 6.1.

3.2 Creating an Attribute

The MOSS formalism is based on properties. Properties are objects and exist by themselves as concepts, independently from any concept or model they can qualify. Thus, it is possible to create properties directly.

There are two kinds of properties: those that qualify an object with a value (e.g. `AGE` of a person), those that link objects to other objects (e.g. `BROTHER` of a person). The former are called *attributes*, and the latter *relations*. OWL calls them respectively *datatype properties* and *object properties*.

3.2.1 Syntax

defattribute *reference &rest option-list* **macro**

reference: reference of the property (symbol, string, multilingual name)

option-list:

option	**arguments**	**explanation**
:class-ref	<class-ref>*	to attach the attribute to a class
:default	<value>	default value
:entry	<function-descriptor>	
:index	<function-descriptor>	specifies entry-point, meaning that values attached to the attribute will produce an index to the object in which the attribute appears. If no argument, uses `make-entry-symbols` to produce the index, otherwise uses specified function where <function-descriptor>::=<arg-list><doc><body> e.g. (:entry (value-list) "Entry for Company name" (list (intern (make-name value-list)))) The entry function must return a list of symbols.
:min	<number>	minimal cardinality
:max	<number>	maximal cardinality
:unique		minimal and maximal cardinality are each 1
:one-of	<value>+	restrictions on the values, that should be taken from a list. Selection is done according to the `:exist` and `:forall` options. Default is `:exists` Only applies when the class is specified (`:class` or `:class-id` options).
:forall		all values should be taken from the one-of list or be of the specified value-type Only applies when the class is specified (`:class` or `:class-id` option) and `:one-of` option is present.
:exists		there should be at least one value taken from the one-of list or of the specified value-type Only applies when the class is specified (`:class` or `:class-id` options) and `:one-of` option is present.
:doc	<doc-string>	documentation

3.2.2 Examples:

Defining attributes directly is not a usual approach. In general attributes are defined as a side effect of the definition of a concept. Nevertheless, it is possible to define attributes independently, either as a preliminary step or a posteriori for adding the attribute to an already defined concept. The useful syntax presents many variations that are described in the following subsections.

Simple Example

The simplest possible syntax is the following:

```
Defining the property age (no options)
? (defattribute "Age")
$T-AGE
```

This creates a number of objects:

* a generic property:

  ```
  ? $T-AGE
  ((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-AGE)) (MOSS::$PNAM (0 (:EN "Age")))
   (MOSS::$ETLS.OF (0 MOSS::$SYS.1)) (MOSS::$INV (0 $T-AGE.OF)))
  ```

  Note that in the internal format the name of the property has been set to `(:EN "Age")` meaning that English has been adopted as the default language.

* an internal name for the property:

  ```
  ? HAS-AGE
  ((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 HAS-AGE)) (MOSS::$PNAM.OF (0 $T-AGE))
   (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
  ```

* a temporary variable containing the identifier of the property:

  ```
  ? _has-age
  $T-AGE
  ```

* an inverse property:

  ```
  ? $T-AGE.OF
  ((MOSS::$TYPE (0 MOSS::$EIL)) (MOSS::$INAM (0 (:EN "IS-AGE-OF")))
   (MOSS::$INV.OF (0 $T-AGE)) (MOSS::$EILS.OF (0 MOSS::$SYS.1)))
  ```

* a name for the inverse property:

  ```
  ? IS-AGE-OF
  ((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 IS-AGE-OF))
   (MOSS::$INAM.OF (0 $T-AGE.OF)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
  ```

* an accessor function (defsettable), useful when writing methods (note that the name of the accessor function is the same as the name of the property):

  ```
  ? HAS-AGE
  used as: (HAS-AGE _jd) ; where _jd is a person
       or: (setf (HAS-AGE _jd) 35)
  ```

Note however that no variable is created for the inverse property, nor any accessor function.

Variations

* The following definitions are equivalent:

```
? (defattribute "Name")          ; using a string
? (defattribute NAME)            ; using a symbol (not recommended, deprecated)
? (defattribute (:en "name"))    ; using a multilingual name
```

  Although the definitions are equivalent, it is better to use strings since they retain the case.

* Defining the property name, with cardinality constraints, e.g. a name can have at least one value and a maximum of three values:

```
? (defattribute "Name" (:min 1)(:max 3))
$T-NAME
```

* Using a default on a generic attribute:

```
? (defattribute "Name" (:default "Dupond" "Durand"))
$T-NAME

? $T-NAME
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-NAME3))
 (MOSS::$PNAM (0 (:EN "Name"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$INV (0 $T-NAME.OF)) (MOSS::$DEFT (0 "Dupond" "Durand")))
```

  The default values (two in this case, are attached directly to the generic property.

* Defining the property name, with default indexing capabilities:

```
? (defattribute "Name" (:entry))
$T-NAME

? $T-NAME
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-NAME))
 (MOSS::$PNAM (0 (:EN "Name"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$INV (0 $T-NAME.OF)) (MOSS::$OMS (0 $FN.193)))
```

  A specific own method is created ($FN.193) and attached to the attribute. Whenever a name will be created, the method will be invoked to create the corresponding indexes.

* Defining the property name, with a documentation string:

```
? (defattribute "Name" (:doc "family name"))
$T-NAME

? $T-NAME
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-NAME))
 (MOSS::$PNAM (0 (:EN "Name"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$INV (0 $T-NAME.OF)) (MOSS::$DOCT (0 (:EN "family name"))))
```

  Note that the string has been transformed into a string of the default language.

* Defining AGE and assigning the attribute to the class PERSON (must be defined)

```
? (defattribute "Age" (:class "Person"))
$T-PERSON-AGE

? $T-PERSON-AGE
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-PERSON-AGE))
 (MOSS::$PNAM (0 (:EN "age"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$IS-A (0 $T-AGE)) (MOSS::$INV (0 $T-PERSON-AGE.OF))
 (MOSS::$PT.OF (0 $E-PERSON)))
```

The newly created property is defined as a local attribute of the class PERSON and gets a different identifier ($T-PERSON-AGE). The attribute is related to the generic $T-AGE attribute by the $IS-A property.

* Defining that one of the names of a student should be taken from a list.

```
? (defattribute "name" (:class "student")
                       (:one-of (:en "judith") (:en "jim") (:en " george")))
$T-STUDENT-NAME

? $T-STUDENT-NAME
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-STUDENT-NAME))
 (MOSS::$PNAM (0 (:EN "name"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$IS-A (0 $T-NAME)) (MOSS::$INV (0 $T-STUDENT-NAME.OF))
 (MOSS::$PT.OF (0 $E-STUDENT))
 (MOSS::$ONEOF (0 (:EN "judith") (:EN "jim") (:EN " george")))
 (MOSS::$SEL (0 :EXISTS)))
```

3.3 Creating a Concept

A concept is an object describing the structure of an individual object that can be considered an instance of this concept. Thus, defining a concept amounts to defining the structure of its instances. One could think of the approach as similar to a class/instance relation in object-oriented languages. However, MOSS does not require that instances of a class have all the properties defined in the concept, or even lets such individuals have properties not present in the concept. Thus, a concept is the description of a typical individual, allowing exceptions to be present in actual individuals.

When specifying a property (attribute or relation) in a concept, MOSS uses the corresponding generic property, creating it if it does not exist. It then creates a local property associated with the new concept.

3.3.1 Syntax

defconcept *reference &rest option-list* **macro**

reference: reference of the concept (symbol, string, multilingual name)

option-list:

option	arguments	explanation
:is-a	<class-ref>*	for defining inheritance (multiple inheritance is allowed)
:att	<attibute-description>	defining an attribute for the current class. All attribute options are allowed except for :class-ref
:rel	<relation-description>	defining an attribute for the current class. All attribute options are allowed except for `:from`.
:one-of	<object-ids>+	specifying the class as a set of objects. Note that the objects must already exist.
:doc	<doc-string>	documentation

The following options are advanced and require knowledge or the internals of MOSS.

option	arguments	explanation
:id	<id>*	id for the class object (if not there, id is synthesized). Reserved for system use.
:do-not-save		Reserved for system use.

3.3.2 Examples

Defining concepts is the normal way of proceeding when defining an ontology. Properties are defined from within the definition of the concept and created if they do not exist.

Simple Example

The simplest example is:

```
(defconcept person)
or
(defconcept "person")
or
(defconcept (:en "person"))
or
(defconcept (:name :en "person"))
```

This defines the concept of person. As such, it does not have any specific properties. However, a number of objects are created:

* the concept object itself:

  ```
  ? $E-PERSON
  ((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-PERSON))
   (MOSS::$ENAM (0 (:EN "Person"))) (MOSS::$RDX (0 $E-PERSON))
   (MOSS::$ENLS.OF (0 MOSS::$SYS.1)) (MOSS::$CTRS (0 $E-PERSON.CTR)))
  ```

  The concept is an instance of the entity metaconcept (MOSS::$ENT). It has a counter with initial value 1 and a radix ($E-PERSON) that will be used to create identifiers of individual persons.

* an index:

```
? PERSON
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 PERSON)) (MOSS::$ENAM.OF (0 $E-PERSON))
 (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

* a temporary variable whose value is the concept identifier:

```
? _person
$E-PERSON
```

Variations

* Adding documentation:

```
? (defconcept "Person"
      (:doc "A PERSON is a human being."))
$E-PERSON
```

* Concept with properties
A concept has normally some properties (attributes or relations):

```
? (defconcept "Person"
   (:att "Name" (:entry))
   (:att "First-Name" (:max 3))
   (:rel "mother" (:to "person") (:max 1))
$E-PERSON

? $E-PERSON
((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-PERSON))
 (MOSS::$ENAM (0 (:EN "Person" :FR "Personne; Individu")))
 (MOSS::$RDX (0 $E-PERSON)) (MOSS::$ENLS.OF (0 MOSS::$SYS.1))
 (MOSS::$CTRS (0 $E-PERSON.CTR))
 (MOSS::$PT (0 $T-PERSON-NAME $T-PERSON2-FIRST-NAME))
 (MOSS::$PS (0 $S-PERSON-MOTHER)) (MOSS::$SUC.OF (0 $S-PERSON-MOTHER)))
```

This creates a concept with a name (indexed), a first name (max cardinality constraint of 3), and a relation linking a person to the person's mother, another person (max cardinality of 1).

* An example with a subclass and documentation:

```
? (defconcept student2 (:is-a person)
   (:doc " A student is a person engaged in a studying program."))
$E-STUDENT
```

Meaning that a student is a person.

3.3.3 Warning

Concepts once defined cannot be redefined. Indeed, it is not a good idea to redefine a concept, because if individuals of this concept exist, it could lead to inconsistencies between the newly defined concept and the existing individuals.

3.4 Creating a Relation

Relations are used for linking objects, in particular the individuals of two concepts. In general relations are created as a consequence of creating concepts. However, they can be created independently. In that case, the concepts must be referenced and must have been defined beforehand for the property to be created. A different design choice could have been to create the missing classes automatically, but the option was dismissed. The adopted approach allows catching spelling mistakes.

3.4.1 Syntax

defrelation *reference &rest option-list* **macro**

reference: reference of the property (symbol, string, multilingual name)
option-list:

option	arguments	explanation
:from	<class-ref>	ref of domain class
:to	<suc-ref>	ref to identify successor class (range)
:min	<number>	minimal cardinality
:max	<number>	maximal cardinality
:unique		minimal and maximal cardinality are each 1
:default	<object-id>	list of object identifiers of objects that can be used as default values for the relation. The objects should be instances of the successor class (range)
:one-of	<value>+	restrictions on the values, that should be taken from a list. Selection is done according to the `:exist` and `:forall` options. Default is `:exists`
:forall		all values should be taken from the one-of list or be of the specified value-type
:exists		there should be at least one value taken from the one-of list or of the specified value-type
:doc	<doc-string>	documentation

The following options are advanced and require knowledge or the internals of MOSS.

option	arguments	explanation
:id	<id>*	id for the relation object (if not there, id is synthesized). Reserved for system use.
:do-not-save		Reserved for system use.

Note that the references in the :from and :to options can be both replaced by the keyword `:any` when one wants to define a generic property that will apply to any object. However it is an error to have only one be assigned the keyword `:any`.

The options `:min, :max, :unique, :var`, and `:doc` apply to generic properties. The options `:one-of, :forall`, and `:exists` do not.

3.4.2 Examples

Simple Example

The simplest syntax for creating a relation is the following (assuming that PERSON is a valid concept):

```
? (defrelation "brother" (:from "person") (:to "person"))
$S-PERSON-BROTHER
```

A number of objects are created:

* An index to the property:

```
HAS-BROTHER
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 HAS-BROTHER))
 (MOSS::$PNAM.OF (0 $S-BROTHER $S-PERSON-BROTHER)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

* The property itself, first generic:

```
$S-BROTHER
((MOSS::$TYPE (0 MOSS::$EPS)) (MOSS::$ID (0 $S-BROTHER))
 (MOSS::$PNAM (0 (:EN "brother"))) (MOSS::$PS.OF (0 MOSS::*ANY*))
 (MOSS::$SUC (0 MOSS::*ANY*)) (MOSS::$ESLS.OF (0 MOSS::$SYS.1))
 (MOSS::$INV (0 $S-BROTHER.OF)) (MOSS::$IS-A.OF (0 $S-PERSON-BROTHER)))
```

* Then, local:

```
$S-PERSON-BROTHER
((MOSS::$TYPE (0 MOSS::$EPS)) (MOSS::$ID (0 $S-PERSON-BROTHER))
 (MOSS::$PNAM (0 (:EN "brother"))) (MOSS::$ESLS.OF (0 MOSS::$SYS.1))
 (MOSS::$IS-A (0 $S-BROTHER)) (MOSS::$INV (0 $S-PERSON-BROTHER.OF))
 (MOSS::$PS.OF (0 $E-PERSON)) (MOSS::$SUC (0 $E-PERSON)))
```

* An inverse property:

```
$S-BROTHER.OF
((MOSS::$TYPE (0 MOSS::$EIL)) (MOSS::$INAM (0 "IS-BROTHER-OF"))
 (MOSS::$INV.OF (0 $S-BROTHER))
 (MOSS::$EILS.OF (0 MOSS::$MOSSSYS)))
```

* An index to the inverse property, generic:

```
$S-BROTHER.OF
((MOSS::$TYPE (0 MOSS::$EIL)) (MOSS::$INAM (0 "IS-BROTHER-OF"))
 (MOSS::$INV.OF (0 $S-BROTHER)) (MOSS::$EILS.OF (0 MOSS::$SYS.1)))
```

* ... and local:

```
$S-PERSON-BROTHER.OF
((MOSS::$TYPE (0 MOSS::$EIL)) (MOSS::$INAM (0 "IS-BROTHER-OF"))
 (MOSS::$INV.OF (0 $S-PERSON-BROTHER)) (MOSS::$EILS.OF (0 MOSS::$SYS.1)))
```

* Internal (temporary variables), containing the property identifiers:

```
_has-BROTHER
$S-BROTHER
_HAS-PERSON-BROTHER
$S-PERSON-BROTHER
```

* Accessor functions (defsettable), useful when writing methods:

```
HAS-BROTHER
used as: (HAS-BROTHER _jd) ; where jd is a person
     or: (setf (HAS-BROTHER _jd) 35)
```

Variations

* The syntax of the defrelation macro is quite flexible.
  It can be positional using references:

```
? (defrelation BROTHER (:from PERSON) (:to PERSON))
? (defrelation (:en "brother") (:from (:en "person")) (:to (:en "person")))
? (defrelation "brother" (:from "person") (:to "person"))
or any mixture of that
? (defrelation BROTHER (:from "person") (:to (:en "person")))
```

* We can impose restrictions on the property (meaning that one can have the French nationality and maybe other ones but no more than 3):

```
(defrelation FRENCH-NATIONALITY (:from PERSON) (:to COUNTRY) (:one-of _france) (:exists)
```

* Generic property (applying to all objects)

```
(defrelation (:en "neighborhood") (:from :any) (:to :any))
```

* Relation one-to-all:

```
(defrelation (:en "zzz") (:from person) (:to universal-class))
```

3.5 Creating an Individual

An individual can be created from a set of properties and values (see Orphans). However, if its type is known, one can use the corresponding concept for that. Note that properties must exist, otherwise they are ignored.

3.5.1 Syntax

defindividual *concept-reference &rest option-list* **macro**

concept-reference: reference to a class (must have been defined)
option-list:

option	arguments	explanation
<attribute-ref>*	<value>+	attribute and values. If the attribute is referenced by a symbol (name), then it must be the internal name of the symbol, e.g. HAS-AGE (do not forget the prefix HAS-)
<relation-ref>*	<id>+	relations and list of object identifiers or variables containing object identifiers. If the relation is referenced by a symbol (name), then it must be the internal name of the symbol, e.g. HAS-BROTHER (do not forget the prefix HAS-). It is however better to use strings.
<relation-ref>*	(:new <class-ref> <individual description>)+)	The second format allows creating local objects while avoiding to use variables
:doc	<doc-string>*	documentation: string or multilingual name. If a string, then will be transformed into a multilingual name using the current language (default is English)
:var	<var-name>	temporary variable name, e.g. _dbb

Simple Example

Defining the person DUPOND-DURAND:

```
(defindividual "person" (HAS-NAME "Dupond" "Durand")("AGE" 23)(:var _dd)
  (:doc "Dupond-Durand is the mayor"))
```

Executing the previous command yields:

* The object:

```
? $E-PERSON.5
((MOSS::$TYPE (0 $E-PERSON)) (MOSS::$ID (0 $E-PERSON.5))
 ($T-NAME (0 "Dupond" "Durand")) ($T-AGE (0 23))
 (MOSS::$DOCT (0 (:EN "Dupond-Durand is the mayor"))))
```

* A temporary variable:

```
? _dd
$E-PERSON.5
```

* Entry-points:

```
? Dupond
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 DUPOND))
 ($T-PERSON-NAME.OF (0 $E-PERSON.5)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))

? Durand
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 DURAND))
 ($T-PERSON-NAME.OF (0 $E-PERSON.5)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

Variations

* An example creating an anonymous address object:

```
? (defindividual person
     (HAS-NAME "Dupond" "Durand")
     ("AGE" 23)
     ("Address"
        (:new "Address" ("street" "10, rue de la Paix")
                        ("city" "Paris")))
     (:var _dd)
     (:doc "Dupond-Durand is the mayor))
```

The specific address object is created as an anonymous object and linked to the individual. It could be considered as a structured value in other formalisms. Note however that the address object could receive a `:var` option, and is a first class object that can be queried.

3.6 Creating an Orphan

An orphan is a classless individual that is created by assembling properties values and links. Note that properties must exist, otherwise they are ignored.

3.6.1 Syntax

The syntax is essentially the same as for creating individuals without the concept reference.

defobject *&rest option-list* **macro**
option-list:

option	arguments	explanation
<attribute-ref>*	<value>+	attribute and values. If the attribute is referenced by a symbol (name), then it must be the internal name of the symbol, e.g. HAS-AGE (do not forget the prefix HAS-)
<relation-ref>*	<id>+	relations and list of object identifiers or variables containing object identifiers. If the relation is referenced by a symbol (name), then it must be the internal name of the symbol, e.g. HAS-BROTHER (do not forget the prefix HAS-). It is however better to use strings.
<relation-ref>*	(:new <class-ref> <individual description>)+)	This second format allows creating local objects while avoiding to use variables
:doc	<doc-string>*	documentation: string or multilingual name. If a string, then will be transformed into a multilingual name using the current language (default is English)
:var	<var-name>	temporary variable name, e.g. _dbb

3.6.2 Example

Defining a red object:

```
? (defobject ("color" "red")(:doc "a red object")(:var _my-object))
```

Executing the previous command yields:

* The object:

```
? $ORPHAN.1
((MOSS::$TYPE (0 MOSS::*NONE*)) (MOSS::$ID (0 $ORPHAN.1)) ($T-COLOR (0 "red"))
 (MOSS::$DOCT (0 (:EN "a red object"))))
```

* A temporary variable:

```
? _my-object
$ORPHAN.1
```

3.7 Creating an Instance Method

An instance method is analogous to a regular method in an object-oriented approach. The method is attached to the concept (class) and applies to the individuals (instances) belonging to this concept. Creating methods is similar to creating Lisp functions with some restrictions.

3.7.1 Syntax

The syntax is essentially the same as for creating individuals without the concept reference.

definstmethod *name selector arg-list &rest body* **macro**
name: name of the method (by convention should start with an = sign)
selector: name of the concept
arg-list: arguments of the method
body: (Lisp) code for the method

3.7.2 Example

The following code defines a method to print an instance of person:

```
? (definstmethod =print PERSON (title)
    (format t "~&   ~A. ~A ~A"
            title (car (HAS-FIRST-NAME)) (car (HAS-NAME))))
```

Defining an individual and applying the method yields:

```
? (defindividual person ("name" "Dupont") ("first name" "John" "Jim")(:var _jjd))
$E-PERSON.8

? (send _jjd '=print "Mr")
   Mr. John Dupont
NIL
```

3.7.3 Method Special Mechanisms

While writing methods one can use the following features:

The special variable `*self*` contains the id of the object that will receive the message.

The special variable `*answer*` contains the answer returned by the last message.

Accessors like `HAS-XXX` can be used as follows:

```
(HAS-XXX obj-id) is equivalent to (send obj-id '=get 'HAS-XXX)
(HAS-XXX) is equivalent to (send *self* '=get 'HAS-XXX)
```

3.8 Creating an Own Method

An own method is attached to a particular object. Creating own method is similar to creating instance methods with the difference that we must specify the object onto which we attach the method.

3.8.1 Syntax

The syntax is essentially the same as for creating individuals without the concept reference.

defownmethod *name selector arg-list &rest body* **macro**
name: name of the method (by convention should start with an = sign)
selector: expression that is used to locate the object to which we want to attach the method
arg-list: arguments of the method

body: (Lisp) code for the method

The selector has a somewhat complex syntax, e.g.

We want attach method to a person named ALBERT, and in case of ambiguity we apply the user-defined function agemax to the resulting list of object ids.

```
(ALBERT HAS-NAME PERSON :select agemax))
```

We want to attach the method to the NAME attribute of the concept PERSON (remember that AGE is the name of a generic property as well as the name of local properties attached to different classes. Thus, :class-ref is used to point to the wanted class).

```
(NAME HAS-PROPERTY-NAME ATTRIBUTE :class-ref PERSON)
```

Warning The selector syntax differs from the previous versions of MOSS: the property name must not include the HAS- prefix like previously:

```
(HAS-NAME HAS-PROPERTY-NAME MOSS-ATTRIBUTE :class-ref PERSON)
```

3.8.2 Example

The following code defines a method attached to the age concept of a person, e.g. for making sure that input values to the age of a person is a number. It implements a semi-predicate.

```
? (defownmethod =xi ("age" HAS-PROPERTY-NAME MOSS-ATTRIBUTE :class-ref PERSON) (data)
    (and (numberp data) data)))
$FN.194
```

* An object has been created for the method (yes, methods are first class objects):

```
? $FN.194
((MOSS::$TYPE (0 MOSS::$FN)) (MOSS::$ID (0 $FN.194)) (MOSS::$MNAM (0 =XI))
 (MOSS::$ARG (0 (DATA))) (MOSS::$CODT (0 ((PRINT DATA) (AND (NUMBERP DATA) DATA))))
 (MOSS::$FNLS.OF (0 MOSS::$SYS.1)) (MOSS::$OMS.OF (0 $T-PERSON-AGE))
 (MOSS::$FNAM (0 $T-PERSON-AGE=S=0=XI)))
```

... and attached to the local property:

```
? $T-PERSON-AGE
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-PERSON-AGE))
 (MOSS::$PNAM (0 (:EN "age"))) (MOSS::$ETLS.OF (0 MOSS::$SYS.1))
 (MOSS::$IS-A (0 $T-AGE)) (MOSS::$INV (0 $T-PERSON-AGE.OF))
 (MOSS::$PT.OF (0 $E-PERSON)) (MOSS::$OMS (0 $FN.194)))
```

3.8.3 Variations

* the selector can take many different forms:

```
("age" "property name" "MOSS attribute" :class-ref "person")
(AGE "property name" "MOSS attribute" :class-ref "person")
```

3.9 Creating a Universal Method

3.9.1 Syntax

Creating universal method is easy since such methods are not attached to any object.

The syntax is essentially the same as for creating individuals without the concept reference.

defuniversal *name arg-list &rest body* **macro**

name: name of the method (by convention should start with an = sign)

arg-list: arguments of the method

body: (Lisp) code for the method

3.9.2 Example

The following example is taken from the MOSS Kernel Library. Any object can receive the =print-error message. Note that body starts with a documentation string.

```
? (defuniversalmethod =print-error (stream msg &rest arg-list)
  "Default method for printing errors - uses LISP format primitive directly.
Arguments:
   stream: printing stream
   msg: format string
   arg-list (rest): args to format string"
  (let ((*moss-output* stream))
    (if (stringp msg)
      (mformat "~%;***Error: ~?" msg arg-list)
      (mformat "~%;***Error:~{ ~A~}" msg))
    ))
```

An object has been created for the method:

The method is created and becomes available for any MOSS object until some own or instance method is created that will shadow it.

3.10 Multilingual Names

Using multilingual names can be quite tricky and must be done carefully. The actual language specification in any package is handled by the global variable `moss::*language*`.

3.10.1 Default Language when Inputting Data

By default the `moss::*language*` variable has value `:en`, meaning that the default language is English. Thus, if nothing is done as was the case in the previous sections, all structures are built using English.

It is possible to change the default language to something else, e.g. French by setting the `moss::*language*` variable to `:fr`.

One cannot set the `moss::*language*` variable to anything. The language code (e.g. `:en` or `:fr` must be one of the code present in the global list `moss::*language-tags*` (e.g. `(:br :en :fr :it :jp :pl)`). Of course, it is possible to add new languages to this list.

Warning: if the input language is :all but language is not specified in the definition, MOSS will add the tag :unknown to the string defining the objects.

3.10.2 Using a Multiple Languages when Inputting Data

In some cases, it is important to build multilingual structures, by accepting several languages simultaneously. In that case the global `moss::*language*` variable must be set to `:all`. The next paragraphs give some examples.

* attribute definition in English and French:

```
? (defattribute (:en "name" :fr "nom"))
$T-NAME

? $T-NAME
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-NAME))
 (MOSS::$PNAM (0 (:EN "name" :FR "nom"))
 (MOSS::$ETLS.OF (0 MOSS::$SYS.1)) (MOSS::$INV (0 $T-NAME.OF)))
```

However, if the generic property was already defined, the new definition does not change it.

```
? (defattribute (:en "name" :pl "nazwa"))
; Warning: reusing previously defined generic attribute HAS-NAME1,
; in package #<Package "COMMON-LISP-USER"> and context 0 ignoring eventual options.
; While executing: MOSS::%MAKE-GENERIC-TP
$T-NAME
```

* concept definition in English and French:

```
? (defconcept (:en "Cook" :fr "Cuisinier; Marmiton"))
;***** === creating class with name: COOK in package #<Package "COMMON-LISP-USER">
$E-COOK
? $E-COOK
((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-COOK))
 (MOSS::$ENAM (0 (:EN "Cook" :FR "Cuisinier; Marmiton"))) (MOSS::$RDX (0 $E-COOK))
 (MOSS::$ENLS.OF (0 MOSS::$SYS.1)) (MOSS::$CTRS (0 $E-COOK.CTR)))
? cook
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 COOK)) (MOSS::$ENAM.OF (0 $E-COOK))
 (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
? cuisinier
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 CUISINIER)) (MOSS::$ENAM.OF (0 $E-COOK))
 (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
? marmiton
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 MARMITON)) (MOSS::$ENAM.OF (0 $E-COOK))
 (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

If the default language is English and we try to define a concept with a French name, then we get an error:

```
? (setq *language* :en)
:EN
```

```
? (defconcept (:fr "essai"))
;***Error (%mln-extract-mln-from-ref) no info for language :EN in
 ref: (:FR "essai")
```

* relation definition in English and French:

  ```
  (defrelation (:en "uncle ; shushu" :fr "oncle; tonton")
               (:from (:en "student ")) (:to (:fr "personne")))
  ```

  Note that the domain and range specifications use one of the references.

* method names (e.g., =test, or =summary) cannot use multilingual names.

3.10.3 Specifying a Language for Output

Output is slightly different.

If the `moss::*language*` variable is set to `:all`, then the system does not know which language to choose. It chooses English by default, and if not possible, a random language or nil.

If the language is specified, it selects the labels in the specified language. If there is no label in this language, it may choose English if present or a random language or nil.

3.10.4 Multilingual Reference

Please refer to the corresponding chapter for details on multilingual names:

3.11 Creating an Ontology

Creating an ontology is done by specifying the package in which the ontology will reside. Several ontologies can exist within the same Lisp environment. All share the concepts and properties of the MOSS system ontology, but each ontology has its own concepts, properties and methods protected from the other ones, meaning that two concepts may exist in different ontologies with different meanings or structures. When MOSS is launched in stand alone (i.e. not part of the OMAS platform) a skeleton ontology is defined in the CG-USER package. One can thus create concepts, properties or methods without calling the defontology macro.

3.11.1 Syntax

The syntax is essentially the same as for creating individuals without the concept reference.

defontology *&rest option-list* **macro**
option-list:

option	arguments	explanation
:language	<tag>	one of the legal languages (:fr, :en,...) or :all meaning that we have multiple languages (default is :en for English)
:package	<keyword>	specifies a package (default: *package*)
:title	<title>	a string title of the ontology (default: "Anonymous Ontology")
:version	<version>	a string, e.g. "2.0" (default: "0.0")

3.11.2 Example

The following declaration creates a new ontology called "FAMILY Test Omtology" in the package :family (created if it does not exist) and it accepts several language, meaning that concepts and properties will have to be defined using a multilingual name.

```
(defontology
  (:title "FAMILY Test Ontology")
  (:package :family)
  (:version "1.0")
  (:language :all)
  )
```

Any user defined new ontology includes the top level MOSS ontology that describes concepts, properties, methods, or entry-points. The MOSS ontology resides in the :moss package. However, its names and entry points are exported for use in other packages. The objects ids are not exported. When creating a new ontology with the defontology macro, an *ontology stub* is created that defines a few needed local classes liked to the MOSS ontology. After that concepts, properties, and methods can be defined as needed. An ontology language (or languages) is not shared with other ontologies. Note that the language of the MOSS ontology is English.

The following examples show how to create a small family ontology. Of course the various concepts, properties or methods can be typed in directly into the interface, but it is better to prepare a file to be loaded once the ontology is ready. Note that MOSS does not allow to define a class twice (a constraint designed to catch spelling mistakes), which requires reloading the system in case of error in a class.

3.11.3 Creating an Ontology in the User Package

When creating an ontology in the user package :cg-user, the ontology stub has already been set by MOSS. Thus, it is not necessary to invoke defontology. For this example we close first the MOSS window so that we can work in the debug window.

Creating and Viewing Concepts

Let us create a concept of PERSON:

```
? (defconcept "Person"
    (:att "name" (:entry))
    (:att "first name" (:max 3))
    (:rel "brother" (:to "Person"))
    (:doc "A PERSON is a human being."))
$E-PERSON
```

We can view the content of the concept structure by evaluating its ID:

```
? $E-PERSON
((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-PERSON))
 (MOSS::$ENAM (0 (:EN "Person"))) (MOSS::$RDX (0 $E-PERSON))
 (MOSS::$ENLS.OF (0 $E-SYS.1)) (MOSS::$CTRS (0 $E-PERSON.CTR))
 (MOSS::$PT (0 $T-PERSON-NAME $T-PERSON-FIRST-NAME))
 (MOSS::$PS (0 $S-PERSON-BROTHER)) (MOSS::$SUC.OF (0 $S-PERSON-BROTHER))
 (MOSS::$DOCT (0 (:EN "A PERSON is a human being."))))
```

It is nicer to ask the concept to print itself:

```
? (send '$E-PERSON '=print-self)

----- $E-PERSON
 CONCEPT-NAME: Person
 RADIX: $E-PERSON
 DOCUMENTATION: A PERSON is a human being.
 ATTRIBUTE : name/Person, first name/Person
 RELATION : brother/Person
 COUNTER: 1
-----
:DONE
```

We could have used the internal temporary[2] variable corresponding to the concept:

```
? (send _PERSON '=print-self)

----- $E-PERSON
 CONCEPT-NAME: Person
 RADIX: $E-PERSON
 DOCUMENTATION: A PERSON is a human being.
 ATTRIBUTE : name/Person, first name/Person
 RELATION : brother/Person
 COUNTER: 1
-----
:DONE
```

Creating an Attribute

Now we can create a new property, say age:

```
? (defattribute "age" (:class-ref "person"))
$T-PERSON-AGE
```

Again we can ask to view the content of the property:

```
? $T-PERSON-AGE
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-PERSON-AGE))
 (MOSS::$PNAM (0 (:EN "age"))) (MOSS::$ETLS.OF (0 $E-SYS.1))
 (MOSS::$IS-A (0 $T-AGE)) (MOSS::$INV (0 $T-PERSON-AGE.OF))
 (MOSS::$PT.OF (0 $E-PERSON)))

? (send _HAS-PERSON-AGE '=print-self)

----- $T-PERSON-AGE
 PROPERTY-NAME: age
-----
:DONE
```

Notice that the nice printing hides a few things to the user. There are other ways of looking at the objects. If one wants to see all the properties of the object including those with no value:

[2]All variables naming concepts and properties will not be kept when objects are saved into the MOSS database.

```
? (send _HAS-PERSON-AGE '=print-all)

----- $T-PERSON-AGE
 ONE-OF:
 VALUE-TYPE:
 VALUE-NOT-TYPE:
 VALUE:
 PROPERTY-NAME: age
 MINIMAL-CARDINALITY:
 MAXIMAL-CARDINALITY:
 VALUE-RESTRICTION:
 DOCUMENTATION:
 TRANSACTION-NUMBER:
 TOMBSTONE:
 SELECTION:
 RESTRICTION-OPERATOR:
-----
:DONE
```

If one wants to see all versions of the object:

```
? (send _HAS-PERSON-AGE '=print-history)
$T-PERSON-AGE
----------
TYPE:                          t0: MOSS::$EPT
IDENTIFIER:                    t0: $T-PERSON-AGE
----- Attributes
PROPERTY-NAME:                 t0: (:EN "age")
----- Relations
IS-A:                          t0: $T-AGE
INVERSE:                       t0: $T-PERSON-AGE.OF
-----Inv-Links
IS-TERMINAL-PROPERTY-LIST-OF:  t0: $E-SYS.1
IS-ATTRIBUTE-OF:               t0: $E-PERSON
----------
:DONE
```

Creating a Sub-Concept

Now let us define the sub-concept of STUDENT:

```
? (defconcept "student" (:is-a "person") (:att "year"))
$E-STUDENT
```

Let us print the content:

```
? (send _student '=print-self)

----- $E-STUDENT
 CONCEPT-NAME: student
 RADIX: $E-STUDENT
 DOCUMENTATION: A PERSON is a human being.
```

```
 ATTRIBUTE : year/student, name/Person, first name/Person, age/Person
 RELATION : brother/Person
 IS-A: Person
 COUNTER: 1
-----
:DONE
```

We can see on the print out that the year attribute is specific to the student, the other attributes or relations being inherited.

Creating a Relation

Now let us add the relation COUSIN to the PERSON concept:

```
? (defrelation "cousin" (:from "person")(:to "person"))
$S-PERSON-COUSIN
```

We can check that it has been done:

```
? (send '$E-PERSON '=print-self)

----- $E-PERSON
 CONCEPT-NAME: Person
 RADIX: $E-PERSON
 DOCUMENTATION: A PERSON is a human being.
 ATTRIBUTE : name/Person, first name/Person, age/Person
 RELATION : brother/Person, cousin/Person
 COUNTER: 1
-----
:DONE
```

Creating Individuals

Some people insist on separating individuals from concept, regarding concepts as part of the ontology and individuals as part of the knowledge base. Thus, concepts have an ontological status and individuals an epistemological status. Although we agree with this position, MOSS allows mixing concepts and individuals.

OK, let us now create a few individuals. First some persons:

```
? (defindividual "person"
        ("name" "Barthès")
        ("first name" "Jean-Paul")
        (:var _jpb))
$E-PERSON.1

? _jpb
$E-PERSON.1

? (send _jpb '=print-self)

----- $E-PERSON.1
 name: Barthès
 first name: Jean-Paul
```

```
-----
:DONE

? (defindividual "person"
       ("name" "Barthès")
       ("first name" "Camille")
       (:var _cb))
$E-PERSON.2

? _cb
$E-PERSON.2

? (send _cb '=print-self)

----- $E-PERSON.2
 name: Barthès
 first name: Camille
-----
:DONE

? (defindividual "student"
    ("name" "Labrousse")
    ("first name" "Sylvie")
    (:var _sl)
    ("cousin" _cb)
    )
$E-STUDENT.1
```

Here we have created a student Sylvie linked to her cousin Camille. Camille is linked back to Sylvie by the inverse link IS-COUSIN-OF. We can see it by printing Camille's history:

```
? (send _cb '=print-history)
$E-PERSON.2
----------
TYPE:                         t0: $E-PERSON
IDENTIFIER:                   t0: $E-PERSON.2
----- Attributes
NAME:                         t0: "Barthès"
FIRST-NAME:                   t0: "Camille"
----- Relations
-----Inv-Links
IS-COUSIN-OF:                 t0: $E-STUDENT.1
----------
:DONE
```

Creating Orphans

Let us now create a new attribute COLOR and describe an object of which we know only its color:

```
? (defattribute "color" (:unique))
$T-COLOR
```

```
? (defobject ("color" "orange") (:var _obj1))
$ORPHAN.1
```

The new object is an orphan, i.e. an object without class:

```
? (send _obj1 '=print-self)

----- $ORPHAN.1
 TYPE: *NONE*
 IDENTIFIER: $ORPHAN.1
 color: orange
-----
:DONE
```

Creating Instance Methods

Now returning to Sylvie. If we print it we get:

```
? (send _sl '=print-self)

----- $E-STUDENT.1
 name: Labrousse
 first name: Sylvie
 cousin: $E-PERSON.2
-----
:DONE
```

The value associated with cousin is the internal identifier of Camille, which is not very user-friendly. We can improve the situation by providing a =summary method[3] to be used whenever a PERSON individual is to be printed.

```
? (definstmethod =summary "person" ()
    (append (has-first-name)(has-name)))
$E-FN.2
```

Now if we print Sophie again:

```
? (send _sl '=print-self)

----- $E-STUDENT.1
 name: Labrousse
 first name: Sylvie
 cousin: Camille Barthès
-----
:DONE
```

Creating Own Methods

We can attach a specific method to any object. Let us define a =summary method for the orphan we have created:

[3]By convention a =summary method must return a list containing a string.

```
? (defownmethod =summary _obj1 ()
    (list (concatenate 'string "Object with color: " (car (has-color)))))
$E-FN.1

? (send _obj1 '=summary)
("Object with color: orange")
```

Note that there is no difference when invoking an own method from invoking an instance method:

```
? (send _jpb '=summary)
("Jean-Paul" "Barthès")
```

Creating Universal Methods

Universal methods apply to any object. One should be careful when defining a universal method. A universal method can be thought of as a default method to be applied when there is no own method nor instance method.

Example:

```
? (defuniversalmethod =position? () :unknown)
$E-UNI.1
? (send _jpb '=position?)
:UNKNOWN
? (send _obj1 '=position?)
:UNKNOWN
? (send _person '=position?)
:UNKNOWN
? (send _has-name '=position?)
:UNKNOWN
```

Of course if really the method is asking for the position of an object, it is silly to ask for the position of the concept of person or of the attribute name...

3.11.4 Creating a Monolingual Ontology in a Specific Package

If we want to create an ontology in a different package, say, an application package called "FAMILY", we must invoke first the defontology macro. Its purpose it to create the specified package if it does not exist already and to build an ontology stub, i.e. the minimal proxy classes representing MOSS classes in the specific package[4].

```
? (defontology  (:title "ONTOLOGIE POUR LA FAMILLE")
      (:language :FR)
      (:package :family)
      (:version "1.0")
      )
; Warning: creating a new package for hosting the ontology: "ONTOLOGIE POUR LA FAMILLE"
version 1.0.
; While executing: MOSS::%MAKE-ONTOLOGY
; Warning: creating an ontology stub in package "FAMILY"
; While executing: MOSS::%MAKE-ONTOLOGY
$E-SYS.1
```

[4]Such proxy classes ($E-FN, $E-SYS, $E-UNI, etc) shadow their MOSS counterpart.

One can then look at the content of the stub:

```
? (send '$E-SYS.1 '=print-self)

----- $E-SYS.1
 SYSTEM-NAME: ONTOLOGIE POUR LA FAMILLE MOSS
 PREFIX: ONTOLOGIE-POUR-LA-FAMILLE-MOSS
 VARIABLE-LIST: *MOSS-SYSTEM*, *ONTOLOGY*
 CREATOR: MOSS
 CREATION-DATE: 16/2/2010
 VERSION: 8.0
 TRANSACTION-NUMBER: 0
 ENTRY-POINT-LIST: ONTOLOGIE-POUR-LA-FAMILLE-MOSS,
     ONTOLOGIE-POUR-LA-FAMILLE-MOSS-SYSTEM, METHOD, UNIVERSAL-METHOD, COUNTER,
     NULL-CLASS, UNIVERSAL-CLASS
 ENTITY-LIST: ONTOLOGIE-POUR-LA-FAMILLE MOSS SYSTEM, METHOD, UNIVERSAL METHOD,
     COUNTER, NULL-CLASS, UNIVERSAL CLASS
-----
:DONE
```

In addition the package in the listener or debug window is changed to "FAMILY" so that we can work in the newly created package. What we can do is similar to what we did in Section 3.11.3 with the difference that the language is now French.

Creating Concepts

Examples:

```
? (defconcept "Personne"
    (:att "nom" (:entry))
    (:att "prénom" (:max 3))
    (:rel "frère" (:to "Personne"))
    (:doc "Une personne est un être humain."))
$E-PERSONNE

? (send _personne '=print-self)

----- $E-PERSONNE
 CONCEPT-NAME: Personne
 RADIX: $E-PERSONNE
 DOCUMENTATION: Une personne est un être humain.
 ATTRIBUTE : nom/Personne, prénom/Personne
 RELATION : frère/Personne
 COUNTER: 1
-----
:DONE

? (defconcept "Étudiant" (:is-a "personne") (:att "année"))
$E-ÉTUDIANT

? (send _étudiant '=print-self)
```

```
----- $E-ÉTUDIANT
 CONCEPT-NAME: Étudiant
 RADIX: $E-ÉTUDIANT
 DOCUMENTATION: Une personne est un être humain.
 ATTRIBUTE : année/Étudiant, nom/Personne, prénom/Personne
 RELATION : frère/Personne
 IS-A: Personne
 COUNTER: 1
-----
:DONE
```

Creating Individuals

Example:

```
? (defindividual "personne"
    ("nom" "Barthès")
    ("prénom" "Jean-Paul")
    (:var _jpb))
$E-PERSONNE.1
? (send _jpb '=print-self)

----- $E-PERSONNE.1
 nom: Barthès
 prénom: Jean-Paul
-----
:DONE
```

3.11.5 Creating a Multilingual Ontology in a Specific Package

The approach is similar to the previous ones with the difference that the language is set to :all and the names must be multilingual.

Ontology

If we want a new ontology package, say :int-family, we use the defontology macro:

```
? (defontology  (:title "INTERNATIONAL FAMILY ONTOLOGY")
      (:language :all)
      (:package :int-family)
      (:version "1.0")
      )
; Warning: creating a new package for hosting the ontology: "INTERNATIONAL FAMILY ONTOLOGY" ve
; While executing: MOSS::%MAKE-ONTOLOGY
; Warning: creating an ontology stub in package "INT-FAMILY"
; While executing: MOSS::%MAKE-ONTOLOGY
$E-SYS.1
```

Otherwise we can simply set the language in a fresh cl-user or cg-user environment:

```
? (setq *language* :all)
:ALL
```

Creating Concepts

Examples:

```
? (defconcept (:en "Person" :fr "Personne; Individu")
    (:att (:en "name" :fr "nom") (:entry))
    (:att (:en "first name; given name" :fr "prénom") (:max 3))
    (:rel (:en "brother" :fr "frère") (:to "Person"))
    (:doc :en "A PERSON is a human being."
          :fr "Une personne est un être humain."))
$E-PERSON
```

Note that we now have a bilingual concept:

```
? (send _person '=print-self)

----- $E-PERSON
 CONCEPT-NAME: Person
 RADIX: $E-PERSON
 DOCUMENTATION: A PERSON is a human being.
 ATTRIBUTE : name/Person, first name/Person
 RELATION : brother/Person
 COUNTER: 1
-----
:DONE

? (with-language :fr (send _person '=print-self))
;Compiler warnings :
;   Undeclared free variable _PERSON, in an anonymous lambda form.
----- $E-PERSON
 CONCEPT-NAME: Personne
 RADIX: $E-PERSON
 DOCUMENTATION: Une personne est un être humain.
 ATTRIBUTE : nom/Personne, prénom/Personne
 RELATION : frère/Personne
 COUNTER: 1
-----
:DONE
```

Note the presence of synonyms in the definitions e.g. "first name; given-name".
Subconcepts:

```
? (defconcept (:fr "Étudiant" :en "Student")(:is-a "personne") (:att "année"))
$E-STUDENT

? $E-STUDENT
((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-STUDENT))
 (MOSS::$ENAM (0 (:FR "Étudiant" :EN "Student"))) (MOSS::$RDX (0 $E-STUDENT))
 (MOSS::$ENLS.OF (0 $E-SYS.1)) (MOSS::$CTRS (0 $E-STUDENT.CTR))
 (MOSS::$IS-A (0 $E-PERSON)) (MOSS::$PT (0 $T-STUDENT-ANNÉE)))

? $T-STUDENT-ANNÉE
```

```
((MOSS::$TYPE (0 MOSS::$EPT)) (MOSS::$ID (0 $T-STUDENT-ANNÉE))
 (MOSS::$PNAM (0 (:UNKNOWN "année"))) (MOSS::$ETLS.OF (0 $E-SYS.1))
 (MOSS::$IS-A (0 $T-ANNÉE)) (MOSS::$INV (0 $T-STUDENT-ANNÉE.OF))
 (MOSS::$PT.OF (0 $E-STUDENT)))

? (send _student '=print-self)

----- $E-STUDENT
 CONCEPT-NAME: Student
 RADIX: $E-STUDENT
 DOCUMENTATION: A PERSON is a human being.
 ATTRIBUTE : année/Student, name/Person, first name/Person
 RELATION : brother/Person
 IS-A: Person
 COUNTER: 1
-----
:DONE
```

Note that we did not specify the language for ”année” and MOSS inserted the :unknown language tag visible in the internal format. Externally however, MOSS prints ”année” as the name of the attribute, which is the only available label it has for it.

Individulats

Individuals properties can be specified using one or the other language, or by a mixture of both:

```
? (defindividual "personne"
    ("nom" "Barthès")
    ("given name" "Jean-Paul")
    (:var _jpb))
$E-PERSON.1

? (send _jpb '=print-self)

----- $E-PERSON.1
 name: Barthès
 first name: Jean-Paul
-----
:DONE

? (with-language :fr (send _jpb '=print-self))
;Compiler warnings :
;   Undeclared free variable _JPB, in an anonymous lambda form.
----- $E-PERSON.1
 nom: Barthès
 prénom: Jean-Paul
-----
:DONE
```

Note: In the case of multilingual names and attributes, access functions like HAS-XXX are limited to the canonical entry of the multilingual name, usually the first English synonym. The main reason is to avoid multiplying the number of such functions. It is always possible to use any synonym to access property values as follows:

```
? (has-first-name _jpb)
("Jean-Paul")

? (send _jpb '=get "given name")
("Jean-Paul")

? (send _jpb '=get "prénom")
("Jean-Paul")
```

But:

```
? (has-given-name _jpb)
> Error: Undefined function HAS-GIVEN-NAME called with arguments ($E-PERSON.1) .
> While executing: "Unknown"
```

Another possible format for stating relation values is the use of the :new keyword as in:

```
? (defindividual "person"
    ("name" "Labrousse")
    ("first name" "Marie-Geneviève")
    ("brother"
     (:new "person" ("name" "Barthès")("first  name" "Pierre-Xavier")))
    (:var _ml))
$E-PERSON.6
```

This automatically creates the Person to which _ml is a brother and link _ml to it.

3.11.6 Creating an Ontology from a Text File

The best way to create an initial ontology is to prepare a text file containing the various definitions and methods. If one puts the file into the applications folder of the MOSS directory, then one can load it from the MOSS window and check its content from that window.

3.11.7 Saving an Ontology into a Database

It is possible to save an ontology into a database (see documentation about Persistency). However, in such cases, several restrictions apply. Among them:

- it is not possible to use macros (not saved into the database)
- global variables must be declared using defvariable
- useful application functions must be declared using deffunction
- accessors like has-XXX cannot be used in application methods (use (g==> XXX) instead)

3.12 Conclusion

This document has presented the syntax of the different macros for building ontologies, knowledge bases and associating methods. It is not a programming manual. Interested readers should consult the MOSS tutorial.

Chapter 4

MOSS Kernel

Contents

Warning: MOSS has a versioning mechanism, i.e. all objects can have different versions in the same environment. The current document is intended for beginners creating a prototype application in their own environment. In so doing, they normally remain in the same context in their own working space. Thus, all the following examples are given assuming a context 0.

4.1 Introduction

The methods documented in this manual implement the basic mechanism of MOSS, and thus belong to its kernel. Some methods apply to all objects; they can be considered as default methods since it is always possible to redefine them at various levels (instance or own). Other methods are attached to the defining classes of the MOSS kernel, i.e. the classes: CONCEPT, RELATION, ATTRIBUTE, INVERSE-LINK, METHOD, UNIVERSAL-METHOD, ENTRY-POINT, COUNTER, SYSTEM; they specify the behavior (or default behavior) of the instances (individuals) of such classes (concepts). Finally some methods deal with the implementation format of the objects; as such they are universal methods, but their action is elementary (e.g. add a value, remove a directed link), and they do not respect the integrity constraints of the PDM model; those are grouped in the section "Formatting methods."

Section 4.2 presents some methods progressively according to what the user intends to do, Section 4.3 presents them by nature (universal methods, methods attached to classes). Section 4.4 presents them alphabetically.

4.2 Convenient Kernel Methods

This section intends to present methods progressively as one would need them to construct a small application. We give their name and a small example. For more details refer to Section 4.4.

4.2.1 Creating Objects

MOSS allows creating objects without first creating the corresponding classes. This may be surprising at first but results from the need to describe unknown objects when captured by the sensors of a robot. Of course to describe an object we need to use attributes. They must be created first.

Creating an Attribute

There are several ways of creating attributes:

- Using defattribute macro
- Using the make-attribute function

For example, we create attributes COLOR, SHAPE and DATE:

```
? (defattribute COLOR)
$T-COLOR
? (m-make-attribute 'SHAPE)
$T-SHAPE
```

Each time the internal id of the property (assigned by the system) is returned. Each name must be unique.

Using one method rather than another is a question of taste and opportunity. When creating interactively, the defattribute macro is convenient, and does not require quoting arguments, when in a program the function is more useful.

In addition to creating a generic attribute in each case, MOSS has created a few other things:

- External names HAS-COLOR, HAS-SHAPE, HAS-DATE
- Temporary variables _has-color, _has-shape, _has-date the values of which are the internal ids
- Accessor functions with the same names as et external names: HAS-COLOR, HAS-SHAPE, HAS-DATE.

Creating Orphans (Classless Objects)

Once attributes or terminal properties are defined it is possible to create objects. This can be done by:

- Using the defobject macro
- Using the moss m-make-object function

For example we create 3 objects:

```
? (defobject ("COLOR" "Orange")(:var _o1))
$ORPHAN.1
? (m-make-object '("SHAPE" "Square")'("color" "Red")'(:var _o2))
$ORPHAN.2

? $ORPHAN.2 ; internal format
((MOSS::$TYPE (0 MOSS::*NONE*)) (MOSS::$ID (0 $ORPHAN.2)) ($T-COLOR (0 "Blue"))
 ($T-DATE (0 3362110553)))
```

The `:var` option allows assigning the internal id to a user-defined external variable, so that it is possible to manipulate it easily. In practice, it is equivalent to do a `setq`, i.e.,

```
? (m-make-object '("SHAPE" "Square")'("color" "Red")'(:var _o2))
```

and

```
? (setq _o2 (m-make-object '("SHAPE" "Square")'("color" "Red"))
```

are equivalent.

Note however that variables associated with the `:var` option must start with an underscore.

Viewing Objects

Objects can be printed using the =print-self method:

```
? (send _o2 '=print-self)

----- $ORPHAN.1
 TYPE: *NONE*
 identifier: $ORPHAN.1
 SHAPE: Square
 COLOR: Red
-----
:DONE

? (send _has-color '=print-self)
----- $T-COLOR
 INVERSE: IS-COLOR-OF
 PROPERTY-NAME: COLOR
-----
:DONE
```

We will see later better ways to display or edit objects.

Creating a Concept (Class)

Creating a concept can be done by:

- Using the defconcept macro
- Using the moss m-make-concept function

Examples

```
? (defindividual ORANGE ("color" "Orange") (:var _orange1))
$E-ORANGE.1
? (m-make-individual 'orange '(:var _orange2))
$E-ORANGE.2

? (m-make-individual _banana '("color" "Yellow") '("shape" "long" "curved") '(:var _banana1))
; Warning: property "shape" is not a property of class "BANANA". We add it anyway.
; While executing: MOSS::%%MAKE-INSTANCE-FROM-CLASS-ID
$E-BANANA.1
? $E-BANANA.1
((MOSS::$TYPE (0 $E-BANANA)) (MOSS::$ID (0 $E-BANANA.1))
 ($T-FRUIT-COLOR (0 "Yellow")) ($T-SHAPE (0 "long" "curved")))
```

On the third example we create a banana with a shape property, that is not part of its concept definition, nor of that of the concept of Fruit. The system however lets us do so, issuing a warning. Note also that the SHAPE attribute is multi-valued. All attributes are multi-valued unless restrictions are imposed on the corresponding properties.

Creating Relations

Once classes are defined it is possible to create relations. This can be done by:

- Using the defrelation macro
- Using the moss m-make-relation function
- Including the property while the class is being defined

Examples

Create a class PERSON (note 2 options :entry, and :min):

```
? (defconcept PERSON (:att NAME (:entry)(:min 1)) (:att FIRST-NAME))
$E-PERSON
```

Create a relationship FOOD between a person and a fruit:

```
? (defrelation FOOD (:from person) (:to fruit))
$S-PERSON-FOOD
? (send _has-person-food '=print-self)

----- $S-PERSON-FOOD
 INVERSE: IS-FOOD-OF
 SUCCESSOR: FRUIT
 PROPERTY-NAME: FOOD
```

```
-----
:DONE
```

Check that it has been added to the class:

```
? (send _person '=print-self)
----- $E-PERSON
 CONCEPT-NAME: PERSON
 RADIX: $E-PERSON
 ATTRIBUTE : NAME/PERSON, FIRST-NAME/PERSON
 RELATION : FOOD/PERSON
 COUNTER: 1
-----
:DONE
```

Check that we can now create a person with food:

```
? (defindividual PERSON ("name" "Dupond")("food" _banana1) (:var _dp))
$E-PERSON.2
? (send _dp '=print-self)
----- $E-PERSON.1
 NAME: Dupond
 FOOD: $E-BANANA.1
-----
"*done*"
```

Or a student?

```
? (defconcept Student (:is-a PERSON))
$E-STUDENT
? (send _student '=print-self)
----- $E-STUDENT
 CONCEPT-NAME: STUDENT
 RADIX: $E-STUDENT
 ATTRIBUTE : NAME/PERSON, FIRST-NAME/PERSON
 RELATION : FOOD/PERSON
 IS-A: PERSON
 COUNTER: 1
-----
:DONE

? (defindividual student (has-food _banana1))
$E-STUDENT.1

? (send '$E-STUDENT.1 '=print-self)

----- $E-STUDENT.1
 FOOD: $E-BANANA.1
-----
:DONE
```

4.2.2 Creating Methods

There is nothing special about a method. It is an object that contains code and that will be used to implement the behavior of other objects.

However, because of orphans (classless objects) there are three kinds of methods, instance-methods, own-methods, and universal-methods.

Instance Methods

Instance methods are like standard methods in most object-oriented languages. They are associated with a class and apply to its instances. They are inherited from super-classes.

Example. Let us define a method that prints "How are you?" (and returns nil) when a person receives the message =HELLO. One can do that by:

- Using the definstmethod macro
- Using the m-make-method function
- Sending a =new message to the model of instance method

Examples

```
? (definstmethod =HELLO PERSON ()
     "polite method #1"
     (princ "How are you?") nil)
$FN.193
? (send _dp '=hello)
How are you?
NIL

? (m-make-method '=HELLO-BIS 'PERSON ()
      '("polite method #2"
        (princ "How are you today?") nil))
$FN.194
? (send _dp '=hello-bis)
How are you today?
NIL

? (send 'moss::$FN '=new '=HELLO-TER 'PERSON ()
     "polite method #3"'((princ "Fine weather, isn't it?") nil))
$FN.195
? (send _dp '=hello-ter)
Fine weather, isn't it?
NIL
```

We have created method objects and can view one as follows:

```
? (send '$FN.195 '=print-object)
----- $FN.195
 TYPE: $FN
 identifier: $FN.195
 METHOD-NAME: =HELLO-TER
 ARGUMENTS: NIL
```

```
 DOCUMENTATION: polite method #3
 CODE: ((PRINC Fine weather, isn't it?) NIL)
 IS-METHOD-LIST-OF: MOSS
 IS-INSTANCE-METHOD-OF: PERSON
 FUNCTION-NAME: $E-PERSON=I=0=HELLO-TER
-----
:DONE
We can check that the method are inherited by students:

? (defindividual STUDENT (HAS-NAME "MoliËre" "Pocquelin")(:var _jbp))
; While executing: MOSS::%MAKE-INSTANCE-FROM-CLASS-ID
$E-STUDENT.1
? (send _jbp '=hello-bis)
How are you today?
NIL
```

Own Methods

A problem arises for objects that are not instances of classes, or for exceptions (i.e., objects that are instances of a class, but that do not behave like the other objects of the same class). For such objects, one can define own-methods, that are attached to the objects directly. To do so one can:

- Using the defownmethod macro
- Using the m-make-ownmethod function
- Sending a =new message to the model of methods ($FN) with an `:own` option.

Examples

```
? (defownmethod =HELLO _dp ()
     "Not so polite method"
   (princ "Grrr...")
   nil)
$FN.196
? (send _dp '=hello)
Grrr...
NIL
? (m-make-own-method '=hello-bis _dp ()
     '("Not so polite method #2"
       (princ "Bad day to you.")
       nil))
$FN.197
? (send _dp '=hello-bis)
Bad day to you.
NIL
? (send 'moss::$FN '=new '=hello-ter _dp ()
        "Not so polite method #3"
        '((princ "Go to hell!")
          nil) '(:own))
$FN.198
? (send _dp '=hello-ter)
```

```
Go to hell!
NIL
```

Universal Methods

Universal methods are methods that apply to any object, i.e., a default method. It can be done by:

- Using the defuniversalmethod macro
- Using the m-make-universal-method function
- Sending a =new message to the model of universal methods ($UNI).

Examples

```
? (defuniversalmethod =BYE () "Default parting method"
      (princ "Bye, nice to have met you...")
       nil)
$UNI.45
? (send _dp '=bye)
Bye, nice to have met you...
NIL
? (send _orange1 '=bye)
Bye, nice to have met you...
NIL
? (send _person '=bye)
Bye, nice to have met you...
NIL
? (m-make-universal-method '=SO-LONG () "Another parting method"
                            '((princ "Bye, bye...") nil))
$UNI.46
? (send _dp '=so-long)
Bye, bye...
NIL
? (send 'moss::$UNI '=new '=CIAO () "a final parting method"
        '((princ "Leaving now...") nil))
$UNI.47
? (send _banana1 '=ciao)
Leaving now...
NIL
```

Universal methods apply without distinction to all objects.

4.2.3 Entry Points

An *entry-point* is an **index** onto an object. It is used to locate objects in the environment. An entry-point is built from the data associated with a terminal property. The simplest way to build entry points is to use the `:entry` option when defining the property.

Example. We already used the :entry option for the property HAS-NAME of a PERSON. Let us see what this implies.

We gave the NAME "Dupond" to a person we created. An entry-point object was automatically created with id the symbol DUPOND.

```
? (send 'dupond '=print-self)
----- $E-PERSON.1
 NAME: Dupond
 FOOD: $E-BANANA.1
-----
:DONE
```

Now what happens if we create a student named "Dupond" ?

```
? (defindividual STUDENT ("name" "Dupond" "Durand")(:var _dd))
$E-STUDENT.2
? (send 'dupond '=print-self)
----- $E-PERSON.1
 HAS-NAME: Dupond
 HAS-FOOD: $E-BANANA.1
-----
----- $E-STUDENT.2
 HAS-NAME: Dupond,Durand
-----
:DONE
```

We can see that the *same entry-point* leads to two different objects that share the same name. Hence, entry-points allow locating objects easily in the environment. Their use takes full significance in queries.

4.2.4 Accessing Values

Accessing values inside an object is done by using the =get universal method.

Example

```
? (send _o2 '=get 'HAS-SHAPE)
("Square")
ou:
? (send _o2 '=get "shape")
("Square")

? (send '$E-PERSON.1 '=get 'HAS-FOOD)
($E-BANANA.1)
? (send '$E-PERSON.1 '=get "  food ")
($E-BANANA.1)
```

Note that the result is always a list, since our properties are multi-valued. Note also that property references can be given as the MOSS name of the property, e.g., HAS-FOOD or has-food or Has-Food, or as a string referencing the property, e.g. " food " or "Food". In the latter case the string will be transformed into the internal name before being processed. Using one or the other is a question of style. Note finally that the case of the data is not important.

Null Values

When =get returns a null value, i.e., NIL, then the meaning is ambiguous. It may mean that the object has no such property or that it is not known that the object has the property.

4.2.5 Adding Data to Objects

Once objects have been created one must be able to add data to them or to remove data. This paragraph deals with adding data.

Adding Values to Attributes

The universal method =add-attribute-values allows adding values to any type of objects.

Examples

```
? (send _o1 '=add-attribute-values 'HAS-COLOR '("Blue" "Green"))
((MOSS::$TYPE (0 MOSS::*NONE*)) (MOSS::$ID (0 $0-0))
 ($T-COLOR (0 "Orange" "Blue" "Green")))

? (send _dp '=add-attribute-values 'has-name '("Dupuis"))
((MOSS::$TYPE (0 $E-PERSON)) (MOSS::$ID (0 $E-PERSON.1))
 ($T-PERSON-NAME (0 "Dupond" "Dupuis")) ($S-PERSON-FOOD (0 $E-BANANA.8)))

? Dupuis
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 DUPUIS))
 ($T-PERSON-NAME.OF (0 $E-PERSON.1)) (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))
```

The strange returned list is in fact the internal format of the objects. One should not be bothered by that. Viewing the internal value of the object is only useful when debugging the system.

Notice on the example that adding the value "Dupuis" to the name of Dupond has automatically created the new corresponding entry-point.

Maximal cardinality can be specified for any given property. However, the user is free to add values past the maximal cardinality. A warning is issued.

When the order of multiple values is important, then the `(:before)` option can be used:

```
? (send _dd '=add-attribute-values 'HAS-NAME '("Dubois") '(:before "Durand"))
((MOSS::$TYPE (0 $E-STUDENT)) (MOSS::$ID (0 $E-STUDENT.2))
 ($T-STUDENT-NAME (0 "Dupond" "Dubois" "Durand")))
```

Adding Links between Objects (Relations)

Links are added by using the =add-relations universal method.

Examples

```
? (send _orange '=add-relations 'HAS-IS-A _FRUIT)
((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-ORANGE))
 (MOSS::$ENAM (0 (:EN "ORANGE"))) (MOSS::$RDX (0 $E-ORANGE))
 (MOSS::$ENLS.OF (0 MOSS::$SYS.1)) (MOSS::$CTRS (0 $E-ORANGE.CTR))
 (MOSS::$PT (0 $T-ORANGE-COLOR $T-ORANGE-SHAPE)) (MOSS::$IS-A (0 $E-FRUIT)))

? (defrelation brother person person)
$S-PERSON-BROTHER
? $S-PERSON-BROTHER
((MOSS::$TYPE (0 MOSS::$EPS)) (MOSS::$ID (0 $S-PERSON-BROTHER))
 (MOSS::$PNAM (0 (:EN "BROTHER"))) (MOSS::$ESLS.OF (0 MOSS::$SYS.1))
```

```
 (MOSS::$IS-A (0 $S-BROTHER)) (MOSS::$INV (0 $S-PERSON-BROTHER.OF))
 (MOSS::$PS.OF (0 $E-PERSON)) (MOSS::$SUC (0 $E-PERSON)))

? (send _dp '=add-relations 'HAS-BROTHER _dd)
((MOSS::$TYPE (0 $E-PERSON)) (MOSS::$ID (0 $E-PERSON.1))
 ($T-PERSON-NAME (0 "Dupond" "Dupuis")) ($S-PERSON-FOOD (0 $E-BANANA.8))
 ($S-PERSON-BROTHER (0 $E-STUDENT.2)))

? (send _dp '=print-self)

----- $E-PERSON.1
 NAME: Dupond,Dupuis
 FOOD: $E-BANANA.8
 BROTHER: $E-STUDENT.2
-----
:DONE
```

We first added an $IS-A link between the class ORANGE and the class FRUIT. Then we created a new relationship BROTHER between persons and told the system that the student Dupond-Durand was a brother of Dupond.

Like attributes, relations may have a maximal cardinality limit, however, the user is allowed to add values past the limit (i.e., cardinality constraints are indicative).

Adding Methods

Adding methods was described in Section **??**.

4.2.6 Removing Data from Objects

A number of methods exist for removing data, whether attached to attributes, relations, or be it objects. In every case the consistency of the internal data structure must be preserved (e.g., entry-points or inverse links).

Removing Attribute Values

One or more values associated with a terminal property may be removed by using the =delete-attribute-values universal method.

Examples

```
? (send _dd  '=print-self)
----- $E-STUDENT.3
 NAME: Dupond,Dubois,Durand
-----
:DONE

?  (send _dd '=delete-attribute-values 'HAS-NAME '("Dupond"))
:DONE
? (send _dd  '=print-self)
----- $E-STUDENT.1
 NAME: Dubois,Durand
```

```
-----
:DONE
? (send 'dupond '=print-self)

----- $E-PERSON.1
 NAME: Dupond,Dupuis
 FOOD: $E-BANANA.1
 BROTHER: $E-STUDENT.3
-----
:DONE
```

We removed the name "Dupond" from the list of names of _dd. The entry-point has been updated.

Note that a minimal cardinality could exist for the property HAS-NAME. If the number of values falls under the limit, a warning message is sent but the values are nevertheless removed.

```
? (send _dd '=delete-attribute-values 'HAS-NAME '("Durand"))
; Warning: The number of values for HAS-NAME in $E-STUDENT.3 is now less than the allowed mini
; While executing: MOSS::*0=DELETE-ATTRIBUTE-VALUES
:DONE
```

Removing Links

The same approach is used to remove links, using the =delete-related-objects universal method.

Example

```
? (send _dp '=delete-related-objects 'HAS-BROTHER (list _dd))
:DONE
? (send _dp '=print-self)
----- $E-PERSON.1
 NAME: Dupond,Dupuis
 FOOD: $E-BANANA.1
-----
:DONE
```

If the property has a minimal cardinality constraint and if the number of links falls under the limit, then a warning message is issued.

Deleting Objects

This can be done by using the =delete universal method.

Example

```
? (send _o1 '=delete)
((MOSS::$TYPE (0 MOSS::*NONE*)) (MOSS::$ID (0)) ($T-COLOR (0)) (MOSS::$TMBT (0 T)))

? (send _o1 '=print-self)
? (send _o1 '=print-self)
reference to $0-0 which does not exist in context 0
> Error: Can't throw to tag :ERROR .
> While executing: MOSS::%%ALIVE?
```

```
> Type Command-. to abort.
See the Restarts? menu item for further choices.
```

When an object is deleted, then all values associated to terminal properties and links associated to structural properties, and inverse links are removed. However the object is not removed from the environment, but a *tombstone* is added for this particular version, as shown in the internal representation returned by the first message. This is to protect other possible versions of the object.

4.2.7 Printing Objects

Several methods can be used for printing objects: =print-self, =print-object, =print-history.

- =print-self uses the class to collect properties to be printed, thus may not print everything
- =print-object prints the actual properties of the object
- =print-history is used to print all versions of a specific object.

Example

```
? (send _dp '=print-self)

----- $E-PERSON.1
 NAME: Dupond,Dupuis
 FOOD: $E-BANANA.1
-----
:DONE

? (send _dp '=print-object)
----- $E-PERSON.1
 TYPE: $E-PERSON
 identifier: $E-PERSON.1
 NAME: Dupond,Dupuis
 FOOD: $E-BANANA.1
 OWN-METHOD: =HELLO OWN/ $E-PERSON.1, =HELLO-BIS OWN/ $E-PERSON.1,
     =HELLO-TER OWN/ $E-PERSON.1
-----
:DONE

? (send _dp '=print-history)

$E-PERSON.1
----------
TYPE:                         t0: $E-PERSON
IDENTIFIER:                   t0: $E-PERSON.1
----- Attributes
NAME:                         t0: "Dupond", "Dupuis"
----- Relations
FOOD:                         t0: $E-BANANA.1
OWN-METHOD:                   t0: $FN.197, $FN.198, $FN.199
BROTHER:                      t0:
-----Inv-Links
```

```
----------
:DONE>
All instances of a given class may be printed using =print-all-instances. However, instances o

? (send _student '=print-all-instances)

1 - $E-STUDENT.1
2 - $E-STUDENT.2
3 - $E-STUDENT.3
:DONE
```

4.2.8 Recovering an Internal Id

Since internal ids are important, there is a way to recover an internal id from an entry point, using the universal method =id.

Example

```
? (send 'DUPOND '=id 'HAS-NAME 'PERSON)
$E-PERSON.1
? DURAND
((MOSS::$TYPE (0 MOSS::$EP)) (MOSS::$ID (0 DURAND)) ($T-PERSON-NAME.OF (0))
 (MOSS::$EPLS.OF (0 MOSS::$SYS.1)))

? (send _dd '=add-attribute-values "nAME" '("Dupond"))
((MOSS::$TYPE (0 $E-STUDENT)) (MOSS::$ID (0 $E-STUDENT.3))
 ($T-PERSON-NAME (0 "Dupond")) ($S-PERSON-BROTHER.OF (0)))

? (send 'DUPONd '=id 'HAS-NAME 'PERSON)
($E-STUDENT.3 $E-PERSON.1)
```

Note that the method can return a single value as a symbol or a list of symbols.

4.2.9 Synonyms

MOSS 6 introduces the possibility of defining synonyms.

Example

```
? (defconcept "town ; city" (:att "size ; number of inhabitants"))
$E-TOWN

? (send 'city '=print-self)
----- $E-TOWN
 CONCEPT-NAME: town
 RADIX: $E-TOWN
 ATTRIBUTE : size /town
 COUNTER: 1
-----
:DONE
```

Inserting semi-columns in a string allows defining any number of synonyms, like town and city in the above example. A concept name being an entry point, the object can be accessed by any of the synonyms. The first name is chosen as the main name, the others are synonyms.

In the same fashion the attribute size has received a synonym, number of inhabitants. Two attribute names have been built, namely HAS-SIZE and HAS-NUMBER-OF-INHABITANTS referring to the same object.

4.2.10 Multilingual Names

MOSS 6 has also been extended to allow multilingual names. By default the language is English. It is specified by the `*language*` global variable set to `:EN`. In practice several names can be given to an object by using a multilingual name.

Example

```
? (defconcept (:name :en "teacher; professor" :fr "enseignant")
    (:att (:name :en "name" :fr "nom") (:entry)))
$E-TEACHER
? (send 'teacher '=print-self)

----- $E-TEACHER
 CONCEPT-NAME: teacher
 RADIX: $E-TEACHER
 ATTRIBUTE : name/teacher
 COUNTER: 1
-----
:DONE
? (send 'enseignant '=print-self)

----- $E-TEACHER
 CONCEPT-NAME: teacher
 RADIX: $E-TEACHER
 ATTRIBUTE : name/teacher
 COUNTER: 1
-----
:DONE
```

Note that we can combine multilingual names with synonyms.

The format for multilingual names is the following:

```
({:name} {<langage-tag> <synonyms>}*)
```

Except for the English tag defined at system level, the other tags can be freely chosen by the designer.

Note however that some languages use extended alphabets and require a UNICODE coding. I recommend using UTF-8.

4.2.11 Multiple Class Belonging (Advanced)

MOSS 7 also has the possibility for an object to belong to several classes.

Example

```
? (defindividual ("teacher" "student")("name" "Albert"))
$E-TEACHER.1

? (send ' $E-TEACHER.1 '=print-history)
$E-TEACHER.1
----------
TYPE:                         t0: $E-TEACHER, $E-STUDENT
IDENTIFIER:                   t0: $E-TEACHER.1, $E-STUDENT.4
----- Attributes
NAME:                         t0: "Albert"
----- Relations
-----Inv-Links
----------
:DONE
```

From the history one can see that the system created two ids for Albert: $E-TEACHER.1 and $E-STUDENT.4, allowing to consider the object as an individual of either concept.

This possibility however should be used with caution.

4.3 Kernel Methods Organized by Type and Class

The following methods implement the basic mechanisms of MOSS, and thus belong to its kernel. Some methods apply to all objects; they can be considered as default methods since it is always possible to redefine them at various levels (instance or own). Other methods are attached to the defining classes of the MOSS kernel, i.e. the classes: ENTITY, ATTRIBUTE, RELATION, INVERSE LINK, METHOD, UNIVERSAL METHOD, ENTRY POINT, COUNTER, SYSTEM; they specify the behavior (or default behavior) of the instances of such classes. Finally some methods deal with the implementation format of the objects; as such they are universal methods, but their action is elementary (e.g. add a value, remove a directed link), and they do not respect the integrity constraints of the PDM model; those are grouped in the section "Formatting methods."

4.3.1 Universal Methods

Each method in the following set can apply to any object in the system; hence they are defined as universal methods.

=add-attribute-values *attribute-name value-list* **Universal method**

adds values associated with an attribute to the object receiving the message. If the attribute has a method building entry points, then all entry points are created accordingly. Checks that the property belongs to the model of the object. If not, then a warning is issued but the property is added nevertheless.

=add-attribute-values-using-id *attribute-id value-list* **Universal method**

is equivalent to the previous method but does less checking and uses the internal attribute identifier directly.

=add-related-objects *relation-name successor-list &rest option-list* **Universal method**

links successor objects to the object receiving the message.

=add-related-objects-using-id *relation-id successor-list* **Universal method**
is equivalent to the previous function but does less checking and uses the internal property identifier directly.

=check **Universal method**
checks an object for compliance with the PDM format. For terminal properties checks that entry points are defined in case they must be. For structural properties, checks for inverse links. For all properties checks that the number of values is between the minimal cardinality and the maximal cardinality when they are specified. The method prints all the defaults it finds in the structure.

=check-cardinality-constraints **Universal method**
when sent to an object returns the list of all properties for which the number of values violates one of the cardinality constraints. I.e., some values are missing, or they are too many.

=clone **Universal method**
makes a copy of any object. The copy is an instance of the same class if the original object was an instance, or else is an orphan (classless object) if the original object was an orphan. All properties and values in the current context are copied in the new object. No checks are done (cardinality, demons). Finally, all entry points are modified to reflect the existence of the new object. Hence a clone has the same entry points as the cloned object.

=clone-from-context **Universal method**
clones an object, importing it from another context. Must be used carefully. Does not import properties that do not exist in the new context. Untested

=delete **Universal method**
deletes the current version of the object from memory, removing the eventual entry-points and cleaning the inverse links. The object is not actually removed from the environment, rather a tombstone is installed in the current context.

=delete-attribute *attribute-reference* **Universal method**
Delete all values associated with a particular attribute and removes the attribute from the object, making it locally unknown rather than empty. Uses the =delete-all method for the bookkeeping.

=delete-attribute-values *attribute-name value-list &rest option-list* **Universal method**
Deletes a list of values from an object, cleaning entry points when necessary.

=delete-attribute-values-using-id *rel-id successor-list &rest option-lis* **Universal method**
Deletes a list of values from an object using the relation id.

=get *property-name* **Universal method**
gets the value list associated to a given property in the current context. If the object has no local value, then its looks in the list of ancestors in a depth first manner; if this does not yield any result then tries to obtain a default value from the property.

=get-default *property-id* **Universal method**
The method is used by =get-id to try to obtain a default value when it could not be obtained from the object itself or from its prototypes. The way to do it is to ask all the classes of the object, then

the property itself. Default so far are only attached to attributes. Default cannot be inherited.

=get-id *property-id* **Universal method**
is analogous to =get but take as an argument the property internal id rather than its external name

=get-properties **Universal method**
Gets the list of all properties (attributes and relations) for a given object. If the object has a class, then the list is obtained from its class; otherwise, properties are returned in a random order.

=has-inverse-properties **Universal method**
returns the list of all inverse property ids. This is used in particular in connection with processing entry points.

=has-properties **Universal method**
returns the list of all properties actually present in the object, with the exception of the property $TYPE and of the inverse properties.

=has-value *property-name* **Universal method**
returns the value list associated to a given property if it is present in the object itself; i.e., does not try to inherit the values, nor looks into default values.

=has-value-id *property-id* **Universal method**
works with the internal property id rather than with its external name.

=inherit-instance *method-name* **Universal method**
is the default inheritance mechanism implementing a lexicographic search (depth first) in case of multiple inheritance. The mechanism invoking this method is disabled in MOSS 7 rendering the method useless. However advanced users can restore the complex inheritance mechanism.

=inherit-isa-instance *method-name* **Universal method**
is the default inheritance mechanism when one tries to inherit an instance method starting with an orphan, and following prototype links to see whther one of the prototypes in the list is an instance of some class. The mechanism invoking this method is disabled in MOSS 7 rendering the method useless. However advanced users can restore the complex inheritance mechanism.

=inherit-own *method-name* **Universal method**
is the default inheritance mechanism (lexicographic depth first) when one looks into the list of prototypes for a local method. The mechanism invoking this method is disabled in MOSS 7 rendering the method useless. However advanced users can restore the complex inheritance mechanism.

=kill-method *method-name* **Universal method**
used to remove a method from the list of objects that can have inherited it and have cached it on their p-list.

=new *&rest dummy-list* **Universal method**
Creates a new instance using the radix contained in the model. Uses =basic-new. Returns the id of a newly created object.

=print-all **Universal method**
prints an object bypassing its print function if present. It uses the method =get-properties and =print-value to print each of them.

=print-error *stream message &rest arg-list* **Universal method**

is the default method that is executed when an error is detected and a =print-error message is sent to an object. It simply prints the message argument using the Lisp format primitive into the *error-output* stream.

=print-history *&key (stream *moss-output*)* **Universal method**
prints the content of an object. For each property prints all values in all contexts from the root of the version graph to the current context.

=print-local-methods *&key (stream *moss-output*)* **Universal method**
prints all methods and documentation associated with the current object.

=print-methods *&key (stream *moss-output*)* **Universal method**
prints all methods and documentation associated with instances of the current object which should be a class. Methods redefined locally are ignored.

=print-object *&key no-inverse (stream *moss-output*)* **Universal method**
prints the content of an object by sending a =print-value message to all its properties including inverse properties except when the option specifies not to print inverse properties.

=print-self *&key (context *context*) (stream *moss-output*)* **Universal method**
prints an object in a nicer format that =print-object.

=replace *property-name value-list* **Universal method**
replaces the value list associated with a given property with the specified value list. Use the methods =delete-all and =add associated with the specific property.

=summary *&rest option-list* **Universal method**
is a default method for returning a list to be used as a summary of any object. The default summary is the internal id of the object!

=unknown-method *method-name* **Universal method**
is the default method for taking care of unknown methods. The default is to print a message stating that the method is not available for the specific object and continuing execution.

=what? *&key (stream t)* **Universal method**
gives a summary of the type of object we are considering, including documentation if available, and the list of ancestors of the model.

4.3.2 Methods Associated with Concepts (Classes)

=add-attribute-default **Instance method**
Adds a default value for a given attribute of a class. The default is added to the ideal, replacing whatever value was present.

=get-instances **Instance method**
gets a list of instances of the corresponding class. Uses the counter to build instance names (not including the ideal).

=instance-name **Instance method**
returns the class name (as a list containing the main name in the current active language as a string).

=instance-summary *individual-id* **Instance method**

returns a list summarizing the object (usually for printing purposes). By default it returns the first value attached to a terminal property. Thus, it is not very useful unless redefined by the user.

=print-all-instances *&rest option-list* **Instance method**
prints instances of a class in abbreviated form or in full format. Default is to print them all using the =summary method. A range can be specified to limit the printing.

=print-instance *instance-id &key (stream *moss-output*)* **Instance method**
Print an instance using model to get list of properties.

=print-name *&key (stream *moss-output*)* **Instance method**
prints the name of the concept into the current active window.

=print-typical-instance *&key (stream *moss-output*)* **Instance method**
prints a typical instance of a class, corresponding to its ideal, i.e., the bearing all the default attributes.

=summary **Instance method**
returns in a list the first name of the concept in the current language.

4.3.3 Methods Associated with Attributes

=add *object-id value-list &rest option-list* **Instance method**
adds a value or a list of new values. Duplicates are NOT discarded. Values are normalized to an internal format before being added (using the =xi method), and then are added at the end of the current list. Cardinality constraints are checked for maximal number of values only. Data can be inserted in front of a specific value if requested.

=check *value-list* **Instance method**
checks that the value list obeys the various restrictions attached to the attribute. Prints eventual errors using mformat. Not sure this method is really useful.

=delete *object-id value &rest option-list* **Instance method**
deletes a single value corresponding to the attribute receiving the message, removing the entry points if any. The value is normalized using the =xi method if any has been defined.

=delete-all *object-id* **Instance method**
deletes all values corresponding to terminal properties of an object, removing the entry points if any.

= format-value *value* **Instance method**
a method normally defined by the user to format a field of data in a special way (i.e., to express a date as an integer). The default method does not do anything and returns the value as it, unless the value is a multilingual name, in which case we extract national canonical part. The method acts as a demon and is called by =add-attribute-values-using-id

=get-instances *aaa* **Own method**
gets all instances of inverse relations by extracting them from the $ETLS property of *moss-system*.

=get-name *value-list* **Instance method**
Returns the name of an instance $ENAM or $PNAM. Calls =instance-name.

=if-added *value object-id* **Instance method**

is a demon for doing some bookkeeping after values have been removed. Is normally called by =add-attribute-values-using-id.

=if-removed *value object-id* **Instance method**
is a demon for doing some book keeping after values have been added. Is normally called by =delete.

=input-value *stream &rest text* **Instance method**
is a demon for querying the user, and eventually doing a dynamic check of the value type. By default print the name of the terminal property, reads whatever is typed back by the user and filters the result by using the =format-value method which acts as a semi-predicate.

=instance-name *&key class* **Instance method**
returns attribute name with associated class name if class keyword is true.

=inverse-id **Instance method**
returns the inverse internal id of the inverse attribute. Needed to build entry points.

=make-standard-entry *data* **Instance method**
Makes standard entry points using make-entry. Returns a list of symbols will be used as ids for the entry points.

=modify-value *object-id value* **Instance method**
is a method normally defined by the user to modify current value attached to a method. The standard default action is to ask the user.

=normalize *value* **Instance method**
Normally normalizes data values - default is to do nothing.

=print *object-id* **Instance method**
is a printing function attached to attributes. Value-list is the set of values to be printed.

=print-value *value-list &rest option-list* **Instance method**
is a printing function attached to an attribute. It prints the property name and the associated values.

=summary **Instance method**
returns the property name.

=xi *value* **Instance method**
obsolete. Replaced by =normalize.

4.3.4 Methods Associated with Relations

=add *object-id suc-list &rest option-list* **Instance method**
adds a successor or a list of new successors. Duplicates are discarded. Successors are added at the end of the current list. Each successor's type is checked and must be allowed by the property. All constraints implemented by means of the =filter and the =if-added method are checked. Whenever they fail, the corresponding successor is not added. Cardinality constraints are checked for maximal value. If too many values are specified, then some of them are discarded. Minimal cardinality is also checked and a warning is eventually issued.

=check *suc-list* **Instance method**

Check that the value list obeys the various restrictions attached to the relation. Prints eventual errors using mformat.

=delete *object-id suc-id* **Instance method**
deletes a link between the two specified objects. It calls the =if-removed demon after the operation.

=delete-all *object-id* **Instance method**
deletes all links corresponding to terminal properties of an object, removing the inverse links.

=filter *successor-id object-id* **Instance method**
is a method normally defined by the user to filter a successor to an object. Criteria are left to the user. The method is applied by =add-related-objects-using-id just before doing the actual link. It is a semi-predicate returning the successor-id if OK, nil otherwise. The default method does not do anything and returns the value as it.

=format-value-list *successor-id* **Instance method**
Deprecated.

=get-instances *&rest option-list* **Own method**
gets all instances of inverse relations by extracting them from the $ESLS property of *moss-system*.

=if-added *successor-id object-id* **Instance method**
is a demon for doing some book keeping after links have been added. Is normally called by =add-related-objects-using-id. The actual bookkeeping is left to the imagination of the user.

=if-removed *&optional old-object-id old-sucessorc-id object-id successor-id* **Instance method**
is a demon for doing some book keeping after links have been removed. Is normally called by =delete. The actual bookkeeping is left to the imagination of the user.

=instance-name *&key class* **Instance method**
returns the property name in a list, showing at what level it was defined if the class key is true.

=instance-summary *object-id* **Instance method**
returns a summary of a structural property using =get-name.

=inverse-id **Instance method**
returns the inverse internal id of the inverse structural property. Needed to build inverse links.

=link *object-id successor-id* **Instance method**
links two objects using the current relation. The link is semantically directed, inverse links are only a facility for navigating in the reverse direction.

=print-value *successor-list &key (stream *moss-output*) header no-value-flag* **Instance method**
Printing function attached to structural properties. Successor-list is the set of successors to be printed. The description printed for each successor is a list obtained by sending the message =summary to each of them in turn.

=summary **Instance method**
returns the property name with the associated class.

=unlink *object-id successor-id* **Instance method**
removes the links between two objects. This is an elementary operation in the sense that no call to the =if-remove daemon is done, contrary to =delete.

4.3.5 Methods Associated with Inverse Links

=delete *object-id successor-id* **Instance method**
deletes a link between object and its successor. No =if-removed daemon is ever called.

=delete-all *object-id* **Instance method**
deletes all existing links between object and its successor. This is used when deleting an entity entirely.

=get-instances *&rest option-list* **Own method**
gets all instances of inverse relations by extracting them from the $EILS property of *moss-system*.

=if-removed *&optional old-object-id old-successor-id object-id successor-id* **Instance method**
Daemon for doing bookkeeping after removing something. Default is to do nothing returning nil.

=instance-name **Instance method**
returns the inverse name qualifying the inverse link.

=inverse-id **Instance method**
returns the property corresponding whose inverse link is the inverse. .

=new **Own method**
Creates a new instance of inverse-link e.g. $S-NAME.OF

=print-value *successor-list &key stream header no-value-flag* **Instance method**
prints a summary of all linked objects.

=summary **Instance method**
returns the inverse property name

=unlink *object-id successor-id* **Instance method**
removes a link between the two objects (same as =delete).

4.3.6 Methods Associated with Entry Points

=get-instances *&rest option-list* **Own method**
gets all instances of entry-points by extracting them from the $EPLS property of *moss-system*.

=id *property classe* **Instance method**
recovers the internal id of an object from its entry-point, associated property, and class. The result can be a list of objects in case of ambiguities.

=merge *application-entry-point* **Instance method**
when reloading application from disk, we must merge app entry-points with entry-points (already active). ***** Uses raw data format.

=print-as-method-for *&key (stream *moss-output*)* **Instance method**
looks at an entry point for a possible name of a method. If so for each object prints the instance method and local method corresponding to the name if any.

=print-self **Instance method**
actually prints all objects associated with a given entry point.

4.3.7 Methods Associated with Methods

=get-instances *&rest option-list* **Own method**
gets all instances of inverse relations by extracting them from the $ESLS property of *moss-system*.

=instance-name **Instance method**
returns the name of the method that received the message.

=print-code **Instance method**
prints the code associated with the given method in a pretty format.

=print-doc **Instance method**
prints the documentation associated with the method object.

=print-self **Instance method**
prints the content of the method: name, documentation, code, etc.

=summary **Instance method**
returns the name of the method.

4.3.8 Methods Associates with Counters

=get-instances **Own method**
get all instances of counters by extracting them from the $SVL property of *moss-system*. Deprecated.

=increment **Instance method**
increases the value of the specific counter by 1.

=new **Own method**
creates a new counter (e.g. $CTR.37) with an initial value of 0.

=summary **Instance method**
returns the value of the counter.

4.3.9 Methods Associated with Systems

=find *entry &optional prop-ref concept-ref* **Instance method**
Extracts entities from KB knowing entry point prop name and concept name.

=get-all-objects **Instance method**
gets all objects present in the system and makes a list of them.

=load-application *application-name* **Instance method**
loads a list of objects from a file-name ((<key> . <value>)*) and reinstalls them. Application objects are first, then we load saved moss objects. Some of the saved moss objects may contain application references (i.e. entry points). Thus, we must merge such objects with the already installed system objects.

=new-classless-key **Instance method**

creates a new key for classless objects

=new-version *aaa* **Instance method**
Adding a new version to the system. Takes the last version of the version-graph adds 1, and forks from current version unless there is an option

```
:from old-branching-context
```

or

```
:from list-of-branching-contexts
```

in which cases contexts are checked for validity before anything is done.

=save-application *&optional (application-name *application-name*)* **Instance method**
Save all the objects of an application including system objects into a file whose name is <application name>.mos
System objects are saved last.

=summary **Instance method**
returns the name of the system.

4.3.10 Methods Associated with Specific Objects

Metaclass

=get-classes *&rest option-list* **Own method**
get all instances of classes by extracting them from the $ENLS property of *moss-system*, own method of $ENT considered as a metaclass.

=set-instances *&rest option-list* **Own method**
Identical to =get-classes.

=get-user-classes *&rest option-list* **Own method**
Deprecated.

=get-user-concepts *&rest option-list* **Own method**
gets all instances of classes by extracting them from the $ENLS property of *moss-system*. All classes in the moss package are removed from the list.

Property: Concept Name

=make-entry *data* **Own method**
is an own-method of the property HAS-CONCEPT-NAME. It is used to create entries for the names of new classes.

Property: Property Name

=make-entry *data* **Own method**
is an own-method of the property HAS-PROPERTY-NAME. It is used to create entries for the names of new properties.

4.4 Kernel Methods (Reference)

The rest of the document contains the list of MOSS Kernel methods, detailing the syntax, arguments, giving examples of use, possible errors.

=add (attribute)

=add *obj-id value-list &rest option-list* **Instance method**

Add the list *value-list* of new values to the object whose identity is *obj-id*. Values can be inserted in a given position by using a `:before` option. Otherwise, values are added at the end of the current list.

Each value is normalized through the =format-value method, and the =if-added method is executed. Whenever they fail the corresponding value is not added. Cardinality constraints are checked for maximal value. If too many values are specified, then a warning is issued *but they are NOT discarded.* Minimal cardinality is also checked and a warning is eventually issued.

We allow duplicate values.

If the property has a method computing entry points, then the corresponding entry points are produced using the =make-entry method.

Option-list:

(`:before` *value*)

When this option is used, then value is normalized using the =xi method. The resulting data is compared with the list of data associated with the current property, and the additional values are added in front of the value specified by the `:before` option.

(`:before-nth` *nth*)

When this option is used the additional successors are added at the nth position in the list.

Examples:

```
? (setq _jim (defindividual PERSON ("NAME" "Jim")))
$E-PERSON.4
? (defindividual PERSON (HAS-NAME "Tom")(:var _tom))
$E-PERSON.5
...
? (send '$T-PERSON-NAME '=add _jim "Julius")
(...($T-PERSON-NAME (0 "Jim" "Julius"))...)
...
? (send '$T-PERSON-NAME '=add _jim "George" (list :before "Julius"))
(...($T-PERSON-NAME (0 "Jim" "George" "Julius"))...)
```

Error or warning messages:

- the value we try to add has not the proper format. This can only be checked when a =format-value method has been associated with the terminal property. The default =format-value method does not check anything.

  ```
  (send '$T-PERSON-NAME '=add _jim '(34))
  ; Warning:  Data 34 has not the right format when trying to associate it with property HAS
  ; While executing: MOSS::$EPT=I=0=ADD
  ```

- note that if we did not define a =format-value method and use a number, the system will declare a severe error, since it cannot build an entry point from a number.

- we try to add too many values, in which case MOSS let the user add the values but issues a warning (remember in our approach the user has the privilege to overrule the system).

```
(send '$T-PERSON-NAME '=add _Jim '(<many values>))
; Warning: Too many values ? we add them anyway...to $E-PERSON.4 for $T-PERSON-NAME
; While executing: MOSS::$EPT=I=0=ADD
```

- The name associated with the `:var` option should be a variable name:

```
? (defindividual PERSON (HAS-NAME "Tommy")(:var tommy))
; Warning: TOMMY is not a valid variable name. We ignore it.
; While executing: MOSS::%%MAKE-INSTANCE-FROM-CLASS-ID
$E-PERSON.5
```

Note:

- This method is essentially used internally by the two universal methods =add-attribute-values and =add-attribute-values-using-id. It is not normally for the user.

=add (relation)

=add *obj-id suc-list &rest option-list* **Instance method**

Add the list *suc-list* of successors to the object whose identity is *obj-id*. Successors can be inserted in a given position by using a :before option. Otherwise, they are added at the end of the current list. Duplicates are discarded.

Each successor is checked through the =filter method, and the =if-added method is executed (daemon). Whenever they fail (i.e. they return nil) the corresponding successor is not added. Cardinality constraints are checked for maximal value. If too many values are specified, then a warning is issued but they are not discarded. Minimal cardinality is also checked and a warning is eventually issued.

Option-list:

(`:before` *suc-id*)

When this option is used the additional successors are added in front of the successor whose id *suc-id* is specified by the `:before` option.

(`:before-nth` *nth*)

When this option is used the additional successors are added at the nth position in the list.

Examples:

```
? (setq _jim (defindividual PERSON ("NAME" "Jim")))
$E-PERSON.4
? (defindividual PERSON (HAS-NAME "Tom")(:var _tom))
$E-PERSON.5
? (defindividual PERSON (HAS-NAME "George")(:var _george))
$E-PERSON.6
...
? (send '$S-PERSON-BROTHER '=add _jim _tom)
(......)
...
? (send '$S-PERSON-BROTHER '=add _jim _george (list :before _tom))
(...($S-PERSON-BROTHER (0 $E-PERSON.6 $E-PERSON.5))...)
```

Error or warning messages:

- the successor is not a PDM object:

```
? (send '$S-PERSON-BROTHER '=add _jim '(34))
; Warning: object 34 is not a PDM object when trying to link it to object $E-PERSON.2
 with property BROTHER
; While executing: MOSS::$EPS=I=0=ADD
```

- we try to add too many values, in which case MOSS let the user add the values but issues a warning (remember in our approach the user has the privilege to overrule the system).

```
(send brother '=add jpb '(<many values>))
; Warning: we are adding more successors than allowed by the cardinality constraint...
 to $E-PERSON.2 for property $S-BROTHER
; While executing: MOSS::$EPS=I=0=ADD
```

Note:

- This function is essentially used internally by the two universal methods =add-related-objects and =add-related-objects-using-id. It is not normally for the user.

=add-attribute-default (concept)

=add-attribute-default *attribute-reference value* **Instance method**

Adds a default value for a given attribute of a class. The default is added to the ideal, replacing whatever value was present.

If called a second time the default value is replaced with the new one.

Examples:

```
? (send _person '=add-attribute-default "sex" "unknown")
((MOSS::$TYPE (0 $E-PERSON)) ($T-PERSON-SEX (0 "unknown")))

? (send _tom '=get 'has-sex)
("unknown")
```

=add-attribute-values *attribute-name value-list &rest option-list* **Universal method**

Adds values associated with an attribute to the object receiving the message. If the attribute has a method building entry points, then all entry points are created accordingly. Checks that the property belongs to the model of the object. If not, then a warning is issued but the property is added nevertheless.

The number of values must be less than the maximal number specified in the attribute HAS-MAXIMAL-CARDINALITY of the property. A warning is issued if the number of values exceeds the maximal number. However all values are added. It is assumed that the user knows what she is doing.

Options:

(`:before` *value*)

When this option is used, then value is normalized using the =xi method. The resulting data is compared with the list of data associated with the current property, and the additional values are added in front of the value specified by the `:before` option.

Example:

```
? (setq _jim (defindividual PERSON (HAS-NAME "Jim")))
$E-PERSON.7
? (send _jim '=add-attribute-values 'HAS-NAME '("john" "joe"))
((moss::$TYPE (0 $E-PERSON)) ($T-PERSON-NAME (0 "Jim" "john" "joe")))
```

MOSS also created the corresponding entry-points JOHN and JOE as names of the person whose internal identifier, $E-PERSON.4, is kept in the variable _jim.

Error messages:

- if the property cannot be found, then a message is issued:

```
? (send _jim '=add-attribute-values 'HAS-MONEY '(20000))
; Warning: Terminal Property HAS-MONEY cannot be found when processing object
 $E-PERSON.7
; While executing: MOSS::*0=ADD-ATTRIBUTE-VALUES
NIL
```

- similarly if the name of the property is ambiguous (i.e. the internal function %extract returns more than 1 value), then an error message is issued and the addition is not done:

```
? (send _jim '=add-attribute-values 'HAS-XXX (list _george))
;***Error: Terminal Property HAS-XXX is an ambiguous name when trying to add values
 to object $E-PERSON.7
```

Warning messages:

- if a value is specified with the `:before-value` option, then it is first normalized with the =xi method if any. If the normalization fails, then a warning is issued and the option is ignored.

```
? (send _jim '=add-tp 'HAS-XXX '("skying") '421)
;***Warning: Value 421 has not the proper format for property HAS-XXX of object
 $E-PERSON.7 and is ignored
```

- when the specified property does not belong to the properties specified for the model of the object or to the already recorded properties for an orphan (in both cases obtained by sending a message =get-properties to the object), then a warning message is issued:

```
? (send jim '=add-tp 'HAS-XXX '("skying"))
;***Warning: Attribute HAS-XXX does not belong to allowed properties for $E-PERSON.7.
 But we add the value anyway
```

Note:

After all the checks on the property name the method =add-attribute-values-using-id is called with essentially the same arguments but with the internal name for the attribute.

=add-attribute-values-using-id (universal)

=add-attribute-values-using-id *attribute-id value-list &rest option-list* **Universal method**

Equivalent to =add-attribute-values but uses the internal property identifier directly.

Options:

(`:before` *value*)

When this option is used, then value is normalized using the =xi method. The resulting data is compared with the list of data associated with the current property, and the additional values are added in front of the value specified by the `:before` option.

Example:

```
? (defattribute SEX (:class PERSON)(:unique))
$T-PERSON-SEX
? (send _jim '=add-attribute-values-using-id '$T-PERSON-SEX "male")
...
```

This is faster than:

```
? (send _jim '=add-attribute-values 'HAS-SEX "male")
```

but the result is the same.

Note:

=add-attribute-values-using-id delegates the work to the property by sending the message

```
(send <att-id> '=add *self* value-list {option-list})
```

=add-related-objects (universal)

=add-related-objects *relation-ref successor-list &rest option-list* **Universal method**

Links successors to a the object receiving the message.

The number of successors must be less than the maximal number specified in the attribute HAS-MAXIMAL-CARDINALITY of the structural property. A warning is issued if the number of values exceeds the maximal number. However all values are added. It is assumed that the user knows what she is doing.

Option:

(`:before` *suc-id*)

When this option is used the additional successors are added in front of the successor whose id *suc-id* is specified by the `:before` option.

Examples:

```
? (setq _jim (defindividual PERSON (HAS-NAME "Jim")))
$E-PERSON.4
? (defindividual PERSON (HAS-NAME "Tom")(:name _tom))
$E-PERSN.5
? (defindividual PERSON (HAS-NAME "George")(:name _george))
$E-PERSON.6
...
?  (send _jim '=add-related-objects 'HAS-BROTHER _tom)
(...($S-PERSON-BROTHER (0 $E-PERSON.5))...)
...
? (send jim '=add-related-objects 'HAS-BROTHER _george (list :before _tom))
(...($S-PERSON-BROTHER (0 $E-PERSON.6 $E-PERSON.5))...)
```

Of course the inverse link is added to tom to allow navigation from tom to jim and from george to jim. However such a link has no semantic meaning and cannot bear constraints.

It is possible to link objects with properties that do not belong to the model of the first object. MOSS complies reluctantly, sending a warning.

```
? (send _jim '=add-related-objects 'HAS-COUSIN _tom)
; Warning: Relation COUSIN does not belong to model of $E-PERSON.4. But we link it anyway...
; While executing: MOSS::*0=ADD-RELATED-OBJECTS
```

Note: After all the checks on the property name the method =add-related-objects-using-id is called with essentially the same arguments but with the internal name of the relation..

=add-related-objects-using-id (universal)

=add-related-objects-using-id *relation-id successor-list &rest option-list* **Universal method**

Links successors to a given object.

Equivalent to previous method =add-related-objects but does less checking and uses the internal property identifier directly. In practice =add-related-objects calls =add-related-objects-using-id.

The number of successors must be less than the maximal number specified in the attribute HAS-MAXIMAL-CARDINALITY of the structural property. A warning is issued if the number of values exceeds the maximal number. However all values are added. It is assumed that the user knows what she is doing.

Option:

(`:before` *suc-id*)

When this option is used the additional successors are added in front of the successor whose id *suc-id* is specified by the `:before` option.

It can be used in sequences like in the following example.

Example:

```
? (setq mother (defrelation MOTHER PERSON PERSON (:unique)))
$S-MOTHER
? (defindividual PERSON (HAS-NAME "Judy")(:var _judy))
$E-PERSON.6

(send _jim '=add-related-objects-using-id mother _judy)
>...
```

This is faster than

```
(send _jim '=add-related-objects-using-id mother _judy)
>...
```

since no check is done on the name of the relation; but the result is the same.

Error messages:

```
? (send _jim '=add-related-objects-using-id mother "judy")
; Warning: object judy is not a PDM object when trying to link it to object $E-PERSON.2
 with property MOTHER
; While executing: MOSS::$EPS=I=0=ADD
```

Note: =add-related-objects-using-id delegates the work to the property by sending the message

```
(send <sp-id> '=add *self* successor-list {option-list})
```

=basic-new (universal)

=basic-new *&rest option-list* **Universal method**

Creates a skeleton of object containing only the $TYPE property in the current context. If object is a class creates an instance otherwise creates an orphan.

Options:

`(:ideal )`

we want to create an ideal instance (with sequence number 0)

Example:

```
? _person
$E-PERSON
? (send _person '=basic-new)
$E-PERSON.1
? $E-PERSON
(($TYPE (0 $ENT)) ($ID (0 $E-PERSON)) ($ENAM (0 (:EN "Person")))
 ($RDX (0 $E-PERSON)) ($ENLS.OF (0 $MOSSSYS)) ($CTRS (0 $E-PERSON.CTR))
 ($PT (0 $T-PERSON-NAME $T-PERSON-AGE $T-PERSON-FIRST-NAME $T-PERSON-SISTER))
 ($PS (0 $S-PERSON-BROTHER)) ($SUC.OF (0 $S-PERSON-BROTHER $S-BOOK-OWNER))
 ($IS-A.OF (0 $E-STUDENT $E-TEACHER $E-BUTCHER $E-RESEARCHER)))
? $E-PERSON.CTR
(($TYPE (0 $CTR)) ($VALT (0 2)) ($CTRS.OF (0 $E-PERSON)) ($SVL.OF (0 $MOSSSYS)))
? $E-PERSON.1
(($TYPE (0 $E-PERSON)) ($ID (0 $E-PERSON.1)))
```

=change-class (universal)

=change-class *new-class-ref &key include-inverse-links* **Universal method**

Takes an object and sets its class to new-class. Copies all properties and values, using generic properties to adapt properties to the class. Copies all values, which may result in constraint violation. The change is done by creating an instance of the new class and transferrring all data and relations to the new instance. If the *include-inverse-links* (key) is true, then relinks inverse links.

The method returns a new id for the object, corresponding to the new class.

Note: Uses =change-class-id

=change-class (universal)

=change-class *new-class-id &key include-inverse-links* **Universal method**

Takes an object and sets its class to new-class. Copies all properties and values, using generic properties to adapt properties to the class. Copies all values, which may result in constraint violation. The change is done by creating an instance of the new class and transferring all data and relations to the new instance. If the *include-inverse-links* (key) is true, then relinks inverse links.

The method returns a new id for the object, corresponding to the new class.

=check (attribute)

=check *successor-list* **Universal method**

Check that the value list attached to the attribute in object obeys the various restrictions attached to the attribute.

Prints eventual errors using mformat.

Example:

```
? (send  '$T-PERSON-NAME '=check '("Jim" "George" "Julius" "34" "john" "joe"))

;warning: too many values (6) for attribute $T-PERSON-NAME, max is 3.
;values: ("Jim" "George" "Julius" "34" "john" "joe")
NIL
```

showing that everything is fine.

Note: =check does not check for missing entry points.

=check (relation)

=check *successor-list* **Universal method**

Checks consistency for a structural property of a given object. Sends a message when the inverse link is not present when it should be there.

Example:

```
? (send _jim '=check)
? (send _jim '=print-history)
$E-PERSON.2
----------
TYPE:                         t0: $E-PERSON
IDENTIFIER:                   t0: $E-PERSON.2
----- Attributes
NAME:                         t0: "Jim", "George", "Julius", 34, "john", "joe"
SEX:                          t0: "male"
----- Relations
BROTHER:                      t0: $E-PERSON.4, $E-PERSON.3
MOTHER:                       t0: $E-PERSON.8
-----Inv-Links
----------
:DONE

? (send $S-PERSON-BROTHER '=check ?($E-PERSON.4 $E-PERSON.3))
($E-PERSON.4 $E-PERSON.3)
```

showing that everything is fine.

Note: =check does not check for missing inverse links.

=check-cardinality-constraints **Universal method**

Sent to an object returns the list of all properties for which the number of values violates one of the cardinality constraints; I.e., some values are missing, or they are too many.

Sort of uninteresting method. Use rather =check on the object.

Example:

The following example shows two violations: one with too many names and the other one with a missing value for sex.

```
? (send _jim '=print-history)
$E-PERSON.2
----------
TYPE:                         t0: $E-PERSON
IDENTIFIER:                   t0: $E-PERSON.2
----- Attributes
NAME:                         t0: "Jim", "George", "Julius", 34, "john", "joe"
SEX:                          t0: "male"
----- Relations
BROTHER:                      t0: $E-PERSON.4, $E-PERSON.3
MOTHER:                       t0: $E-PERSON.8
-----Inv-Links
----------
:DONE

? (send jim '=check-cardinality-constraints)
($T-PERSON-NAME)

? (send '$T-PERSON-NAME '=print-self)

----- $T-PERSON-NAME
 INVERSE: IS-NAME-OF
 PROPERTY-NAME: NAME
 MINIMAL-CARDINALITY: 1
 MAXIMAL-CARDINALITY: 3
-----
:DONE
```

=clone (universal)

=clone **Universal method**

Makes a copy of any object. The copy is an instance of the same class for instances, or else is an orphan (classless object) if the original object was an orphan.

All properties and values in the current context are copied into the new object.

Finally, all entry points are modified to reflect the existence of the new object. Hence a clone has the same entry points as the cloned object.

Example:

```
? (send _dp '=print-history)

$E-PERSON.1
----------
HAS-TYPE:                 t0: $E-PERSON
-----
HAS-NAME:                 t0: Dupond, Dupuis
-----
HAS-FOOD:                 t0: $E-BANANA.1
HAS-OWN-METHOD:           t0: $FN.78, $FN.79, $FN.80
HAS-BROTHER:              t0: $E-STUDENT.1
-----Inv-Links
----------
:DONE
? (send dp '=clone)
$E-PERSON.6
? (send '$E-PERSON.6 '=print-history)

$E-PERSON.6
----------
HAS-TYPE:                 t0: $E-PERSON
-----
HAS-NAME:                 t0: Dupond, Dupuis
-----
HAS-FOOD:                 t0: $E-BANANA.1
HAS-OWN-METHOD:           t0: $FN.78, $FN.79, $FN.80
HAS-BROTHER:              t0: $E-STUDENT.1
-----Inv-Links
----------
:DONE
? (send 'DUPOND '=print-object)
----- DUPOND
 HAS-TYPE: $EP
 IS-NAME-OF: $E-PERSON.1, $E-PERSON.6
 IS-ENTRY-POINT-LIST-OF: MOSS
-----
NIL

? cloning an orphan:
? (send _O2 '=print-history)
```

```
$0.2
----------
TYPE:                    t0: *NONE*
IDENTIFIER:              t0: $0.2
-----
SHAPE:                   t0: Square
COLOR:                   t0: Red
NAME:                    t0: my orange
-----
-----Inv-Links
----------
:DONE
? (send _O2 '=clone)
$0.4
? (send '$0.4 '=print-history)

$0.4
----------
TYPE:                    t0: *NONE*
IDENTIFIER:              t0: $0.4
-----
SHAPE:                   t0: Square
COLOR:                   t0: Red
NAME:                    t0: my orange
-----
-----Inv-Links
----------
:DONE
? (send 'MY.ORANGE '=print-self)
----- $0.2
 TYPE: *NONE*
 SHAPE: Square
 COLOR: Red
 NAME: my orange
-----
----- $0.4
 TYPE: *NONE*
 SHAPE: Square
 COLOR: Red
 NAME: my orange
-----
:DONE
```

=delete (attribute)

=delete *object-id value* **Universal method**

Deletes a value corresponding to a terminal property of an object, removing the entry points if any.

Before being removed value is transformed into its internal format by invoking the =xi method.

Invokes the =if-removed method on the current property, to do whatever bookkeeping must be done.

Example:

```
? (send '$T-PERSON-NAME '=delete pete "Petrus")
; Warning: The number of values for $T-NAME in $E-PERSON.11 is now less
 than the allowed minimum of 1
; While executing: MOSS::$EPT=I=0=DELETE
((MOSS::$TYPE (0 $E-PERSON)) ($T-PERSON-NAME (0)) ($S-PERSON-SISTER (0)))
? (send pete '=print-self)
----- $E-PERSON.11
-----
:DONE
```

Error messages: When there is a =xi formatting method and the value has not the proper format an error message is printed:

```
? (send 'XXX '=delete jim 45)
;***Error: Trying to erase value 45 from object $E-PERSON.12 for property XXX.
 Value has a wrong format
```

=delete (inverse-link)

=delete *object-id suc-id* **Instance method**

Deletes a link between an object and its predecessor. Same as the =delete method for relations, but applies to inverse links (i.e., inverse relations).

=delete (relation)

=delete *object-id suc-id* **Universal method**

Deletes a link between an object and its successor.

Invokes the =if-removed method on the direct property for each predecessor, to do whatever book-keeping must be done.

Example:

```
? (defindividual PERSON (HAS-NAME "Mary")(:var _sis))
$E-PERSON.10
? (defrelation SISTER PERSON PERSON)
$S-PERSON-SISTER
? (defindividual PERSON (HAS-NAME "Petrus")(:var _pete)(HAS-SISTER _sis))
$E-PERSON.11
? (send _pete '=print-self)

----- $E-PERSON.11
 NAME: Petrus
 SEX: unknown
 SISTER: $E-PERSON.10
-----
:DONE

? (send '$S-PERSON-SISTER '=delete _pete _sis)
((MOSS::$TYPE (0 $E-PERSON)) (MOSS::$ID (0 $E-PERSON.11))
 ($T-PERSON-NAME (0 "Petrus")) ($S-PERSON-SISTER (0)))
? (send _pete '=print-self)

----- $E-PERSON.11
 NAME: Petrus
 SEX: unknown
-----
:DONE
```

=delete (universal)

=delete **Universal method**

When sent to an object, deletes the current version of the object from memory, removing the eventual entry-points and cleaning the inverse links. Since the object cannot be physically removed because of the versioning mechanism, we add a tombstone to indicate its death in the current context.

Example:

```
? $E-PERSON.2
(($TYPE (0 $E-PERSON)) ($T-NAME (0 "Jim" "john" "joe"))
 ($S-BROTHER (0 $E-PERSON.4 $E-PERSON.3)) ($S-MOTHER (0 $E-PERSON.5)))
? (send jim '=delete)
(($TYPE (0 $E-PERSON)) ($T-NAME (0)) ($S-BROTHER (0)) ($S-MOTHER (0)) ($TMBT (0 T)))
```

One cannot use =print-history here since the object has been erased. We cheat by using an internal moss function.

```
? (moss::%pep _jim)
$E-PERSON.2
----------
TYPE:                          t0: $E-PERSON
IDENTIFIER:                    t0: $E-PERSON.2
----- Attributes
NAME:                          t0:
SEX:                           t0:
TOMBSTONE:                     t0: T
----- Relations
BROTHER:                       t0:
MOTHER:                        t0:
-----Inv-Links
----------
:DONE
```

=delete-all (attribute)

=delete-all *object-id* **Instance method**

Deletes all values corresponding to attributes of an object, removing the entry points if any.

Invokes the =if-removed method on the current property for each value, to do whatever bookkeeping must be done.

Note: This method is equivalent to doing a loop with the =delete method on the property but is faster.

=delete-all (inverse link)

=delete-all *object-id* **Instance method**

Deletes all links corresponding to an inverse link of an object.

Invokes the =if-removed method on the direct property for each predecessor, to do whatever bookkeeping must be done.

Does not check for minimal cardinality constraints.

Note: This method is equivalent to doing a loop with the =delete method on the property but is faster.

=delete-all (relation)

=delete-all *object-id* **Instance method**

Deletes all links corresponding to relations of an object, removing the inverse links.

Invokes the =if-removed method on the current property for each successor, to do whatever book-keeping must be done.

Note: This method is equivalent to doing a loop with the =delete method on the property but is faster.

=delete-all-successors (universal)

=delete-all-successors *rel-ref* **Universal method**

Deletes all successors to a given entity. Issues a warning when the number of related objects falls below the min cardinality level.

=delete-attribute (universal)

=delete-attribute *attribute-reference* **Universal method**

Delete all values associated with a particular attribute and removes the attribute from the object, making it locally unknown rather than empty. Uses the =delete-all method for the bookkeeping.

Error messages: Same as =delete-all.

=delete-attribute-values (universal)

=delete-attribute-values *attribute-reference value-list* **Universal method**

Delete some values associated with an attribute. If a value is not present is done and no message is sent..

Delegates the work to the =delete method associated with the specified property. See =delete for more explanations and error messages.

Error messages:

```
Attribute HAS-AGE cannot be found.
Attribute HAS-AGE is an ambiguous name.
Something wrong with attribute HAS-AGE while trying to delete 23 from $E-PERSON.2.
```

Warning messages:

```
The number of values for HAS-NAME in $E-PERSON.4 is now less the allowed minimum of 1.
```

=delete-attribute-values-using-id (universal)

=delete-attribute-values-using-id *attribute-id value-list* **Universal method**

Basically the same thing as =delete-attribute-values, but uses id rather than reference.

=delete-related-objects (universal)

=delete-related-objects *relation-reference successor-list* **Universal method**

Deletes a list of successors from an object

Error messages:

```
Relation HAS-NIECE cannot be found
Relation HAS-NIECE is an ambiguous name
Something wrong with relation HAS-NIECE while trying to delete $E-PERSON.6
 from $E-PERSON.10
```

Note: Delegates the work to the =delete method once the internal id of the specified property has been established. See =delete for more explanations and error messages.

=delete-related-objects-using-id (universal)

=delete-related-objects-using-id *relation-id successor-list* **Universal method**

Same as =delete-related-objects but uses property id rather than reference.

=filter (relation)

=filter *successor-id object-id* **Own method**

A method normally defined by the user to filter a successor to an object. Criteria are left to the user. The method is applied by =add-related-objects-using-id just before doing the actual link. It is a semi-predicate returning the successor-id if OK, NIL otherwise.

The default method does not do anything and returns the value as it.

Two arguments must be given: the id of the successor that is a candidate to being linked, and the id of the object it should be linked to. The method can thus access the two objects if some fancy reasoning is required.

Example: The =filter method could be used to check for the sex of a successor of a person for the brother value. Moreover a full expert system could be called to decide whether an object can be effectively linked to another one using the corresponding structural property.

Let us show a simple case checking the sex of a person before allowing to make a link with the brother property. If the sex is not specified, we assume that the link can be made.

```
(defownmethod =filter '$S-PERSON-BROTHER (suc-id obj-id)
  "Filter persons with sex other than male when specified"
   (let ((sex (car (HAS-SEX suc-id))))
      (if (or (null sex)
               (equal sex "m"))
          suc-id
          (progn
             (warn
              "successor ~A is not male (~A) and cannot be
               declared a brother of object ~A"
               suc-id sex obj-id)
              nil))))
$FN.81

? (send '$E-PERSON.6 '=print-self)
----- $E-PERSON.6
 NAME: Mary
 SEX: F
-----
:DONE

? (send '$S-PERSON-BROTHER '=filter '$E-PERSON.6  _dp)


; Warning: successor $E-PERSON.6 is not male (F) and cannot be declared a brother of
 object $E-PERSON.1
; While executing: MOSS::$S-BROTHER=0=0=FILTER
NIL
```

=find (system)

=find *entry &optional property-reference concept-reference* **Instance method**

Extract entities from KB knowing entry point, and eventually prop name and concept name. This is a low-level method and is not really meant to replace a query system.

Example:

```
? (defindividual PERSON (HAS-NAME "Durand") (HAS-FIRST-NAME "Jean")(:var _jd))
$E-PERSON.9
? (defindividual PERSON (HAS-NAME "Dubois" "Durand") (HAS-FIRST-NAME "Albert")(:var _add))
$E-PERSON.10
? (defindividual STUDENT (HAS-NAME "Durand" "Rufus") (HAS-FIRST-NAME "JÈrÙme")(:var _jdr))
$E-STUDENT.2
? (send *system* '=find 'durand)
($E-PERSON.9 $E-PERSON.10 $E-STUDENT.2)
? (send moss::*moss-system* '=find 'durand "name")
($E-PERSON.13 $E-PERSON.14 $E-STUDENT.5)
? (send moss::*moss-system* '=find 'durand "name" "student")
$E-STUDENT.5
```

=format-value (attribute)

=format-value *value* **Own method**

A method normally defined by the user to format a field of data in a special way (i.e., to express a date as an integer). The default method does not do anything and returns the value as it. The method acts as a demon and is called by =add-attribute-values-using-id. It can be used to check the input type of the object and to request a correction from the user for interactive sessions.

In the default method, when value is a multilingual name, we extract national canonical part otherwise we return the value unchanged.

The method is called by the system each time a value associated with a terminal property is added to the system.

Example:

```
(defownmethod =format-value _has-sex (value)
   "Checks the format of the value for sex: allowed values are f, female, m, male ~
    expressed as strings. They are normalized to the short form f or m."
        (cond
         ((member value '("f" "female") :test #'equal)
          "f")
         ((member value '("m" "male") :test #'equal)
          "m")
         (t (warn
            "value for sex must be f or m expressed as strings.") nil)))

? (send _has-sex '=format-value "female")
"f"

? (send _has-sex '=format-value "m")
"m"

? (send _has-sex '=format-value "g")
; Warning: value for sex must be f or m expressed as strings.
; While executing: MOSS::$T-SEX=O=O=FORMAT-VALUE
NIL
```

=get (universal)

=get *property-reference* **Universal method**

Obtains a value for the specified property in the current context. Returns the value-list associated with the current property or inherited from a prototype, or from a default value.

Always return a list of values since all properties are multi-valued.

Example:

```
? (send _dp '=get 'HAS-NAME)
("Dupond" "Dupuis")
```

Error messages:

- Error occurs when the specified property is not part of the object, in which case an empty list is returned.

```
? (send _dp '=get 'HAS-MONEY)
; Warning: in =get HAS-MONEY is not a property of object ($E-PERSON.1) in context 0
; While executing: MOSS::*0=GET
NIL
```

Note: Uses the =get-id method once the id of the property id has been established.

=get-all-objects **Instance method**

gets all objects present in the system and makes a list of them.

=get-classes (concept)

=get-classes *property-reference* **Own method**

Gets all instances of classes by extracting them from the $ENLS property of `*moss-system*`, own method of $ENT considered as a metaclass.

Example:

```
? (send 'moss::$ENT '=get-classes)
(MOSS::$CTR MOSS::$FN MOSS::$UNI MOSS::$SYS MOSS::$ENT MOSS::$EPR MOSS::$EPS
MOSS::$EPT MOSS::$EP MOSS::$EIL MOSS::*NONE* MOSS::*ANY* MOSS::$DOCE
MOSS::$ENDCE MOSS::$FRDCE MOSS::$CNFG MOSS::$USR MOSS::$QHDR MOSS::$QRY
MOSS::$QSTE MOSS::$QFSTE MOSS::$QSSTE MOSS::$QCXT MOSS::$SCXTE MOSS::$QHDE
MOSS::$CVSE MOSS::$ARGEMOSS::$OBRGE MOSS::$ACTE MOSS::$MARGE MOSS::$HAPE
MOSS::$PAPE MOSS::$WAPE $E-ORANGE $E-FRUIT $E-APPLE $E-BANANA $E-PERSON
$E-STUDENT $E-TOWN $E-TEACHER)
```

=get-default (universal)

=get-default *prop-id* **Universal method**

The method is used by =get-id to try to obtain a default value when it could not be obtained from the object itself or from its prototypes. The way to do it is to ask all the classes of the object, then the property itself. Default so far are only attached to attributes. Defaults cannot be inherited.

Example:

```
? (send '$e-person.1 '=get-default '$T-person-sex)
("unknown")
```

=get-documentation *&key no-summary (lead "") final-new-line* **Universal method**

Get an object documentation as a string or sorry message.

Optional key arguments:

no-summary (key): if t doest not print a leading summary of the object

lead (key): string, if there prints it before printing doc (default null string)

final-new-line (key): if t add a new line at the end of the string.

Returns a string.

Example:

```
? (send 'moss::$ent '=get-documentation)
"
CONCEPT : *sorry no documentation available*"
```

=get-id (universal)

=get-id *property-id* **Universal method**

Obtains a value for the specified property in the current context. Returns the value-list associated with the current property or inherited from a prototype, or from a default value.

Always return a list of values since all properties are multi-valued. *Algorithm:*

1. quits if property-id is nil
2. tries to get a value locally by using the internal
3. If no ancestors for the object tries to get a default value by sending the message:

   ```
   (send prop-id '=get-id '$DEFT)
   ```

   unless property is $DEFT (to avoid infinite loops).
4. when there are ancestors, tries to get a value depth first.
5. if there were ancestors, but we could not find a value, we try the default value as in case 3.

Example:

```
? (send dp '=get-id '$T-PERSON-NAME)
("Dupond" "Dupuis")
```

In practice HAS-NAME is declared implicitly as an accessor function that plays the same role.

```
? (HAS-NAME _dp)
("Dupond" "Dupuis")
```

=get-instances (attribute)

=get-instances *prop-id* **Own method**

Gets all instances of attributes by extracting them from the $ETLS property of `*moss-system*`.

Example:

```
? (send 'moss::$EPT '=get-instances)
(MOSS::$VALT MOSS::$CNAM MOSS::$XNB MOSS::$TMBT MOSS::$DOCT MOSS::$ARG MOSS::$REST
 MOSS::$CODT MOSS::$FNAM MOSS::$MNAM MOSS::$UNAM MOSS::$SNAM MOSS::$PRFX MOSS::$SVL
 MOSS::$CRET MOSS::$DTCT MOSS::$VERT MOSS::$ENAM MOSS::$LBLT MOSS::$RDX MOSS::$ONEOF
 MOSS::$PNAM MOSS::$MINT MOSS::$MAXT MOSS::$VRT MOSS::$SEL MOSS::$OPR MOSS::$TPRT
 MOSS::$NTPR MOSS::$INAM MOSS::$TYPE MOSS::$DEFT MOSS::$OBJNAM MOSS::$TIT
 MOSS::$VNBT MOSS::$CFNM MOSS::$USRNAM MOSS::$ACRT MOSS::$CVRS MOSS::$REF MOSS::$ID
 MOSS::$QNAM MOSS::$QXPT MOSS::$QUXT MOSS::$QANT MOSS::$STNAM MOSS::$XPLT
 MOSS::$SOBT MOSS::$DATT MOSS::$DOMT MOSS::$GOLT MOSS::$QT MOSS::$ACT MOSS::$PRMT
 MOSS::$MSGT MOSS::$ANST MOSS::$RSLT MOSS::$QRYT MOSS::$WLT MOSS::$OWS MOSS::$IWS
 MOSS::$TDO MOSS::$AGT MOSS::$FLT MOSS::$ARNAM MOSS::$ARKT MOSS::$ARVT MOSS::$ARQT
 MOSS::$CREFT MOSS::$ANAM MOSS::$OPRT MOSS::$T-MOSS-OBJECT-ARGUMENT-ARG-KEY
 MOSS::$T-HOW-ACTION-OPERATOR MOSS::$T-PRINT-ACTION-OPERATOR
 MOSS::$T-WHAT-ACTION-OPERATOR $T-COLOR $T-SHAPE $T-DATE $T-ORANGE-COLOR
 $T-ORANGE-SHAPE $T-FRUIT-COLOR $T-TASTE $T-FRUIT-TASTE $T-APPLE-DATE $T-BANANA-DATE
 $T-NAME $T-PERSON-NAME $T-FIRST-NAME $T-PERSON-FIRST-NAME $T-SIZE $T-TOWN-SIZE
 $T-SEX $T-PERSON-SEX)
```

=get-instances (concept)

=get-instances *&key min max ideal* **Instance method**

Gets a list of instances of the corresponding class. Uses the counter to build the instance names (including the ideal on option).

Example:

```
? (send _person '=get-instances)
($E-PERSON.1 $E-PERSON.2 $E-PERSON.3 $E-PERSON.4 $E-PERSON.5 $E-PERSON.6 $E-PERSON.7
 $E-PERSON.8 $E-PERSON.9 $E-PERSON.10 $E-PERSON.11 $E-PERSON.12 $E-PERSON.13
 $E-PERSON.14)
? (send _person '=get-instances :min 3 :max 9)
($E-PERSON.3 $E-PERSON.4 $E-PERSON.5 $E-PERSON.6 $E-PERSON.7 $E-PERSON.8 $E-PERSON.9)
```

=get-instances (concept)

=get-instances **Own method**

Gets all instances of concepts by extracting them from the $ENLS property of `*moss-system*`.

Example:

```
? (send 'moss::$ent '=get-instances)
(MOSS::$CTR MOSS::$FN MOSS::$UNI MOSS::$SYS MOSS::$ENT MOSS::$EPR MOSS::$EPS
 MOSS::$EPT MOSS::$EP MOSS::$EIL MOSS::*NONE* MOSS::*ANY* MOSS::$DOCE MOSS::$ENDCE
 MOSS::$FRDCE MOSS::$CNFG MOSS::$USR MOSS::$QHDR MOSS::$QRY MOSS::$QSTE MOSS::$QFSTE
 MOSS::$QSSTE MOSS::$QCXT MOSS::$SCXTE MOSS::$QHDE MOSS::$CVSE MOSS::$ARGE
 MOSS::$OBRGE MOSS::$ACTE MOSS::$MARGE MOSS::$HAPE MOSS::$PAPE MOSS::$WAPE $E-ORANGE
 $E-FRUIT $E-APPLE $E-BANANA $E-PERSON $E-STUDENT $E-TOWN $E-TEACHER)
```

=get-instances **Own method**

Gets all instances of concepts by extracting them from the $ETLS property of *moss-system*. Deprecated.

=get-instances (entry point)

=get-instances **Own method**

Get all instances of entry-points by extracting them from the $EPLS property of `*moss-system*`.

Example:

```
? (send '$EP '=get-instances)
(MOSS COUNTER HAS-VALUE IS-VALUE-OF HAS-COUNTER-NAME IS-COUNTER-NAME-OF
 HAS-TRANSACTION-NUMBER IS-TRANSACTION-NUMBER-OF HAS-TOMBSTONE IS-TOMBSTONE-OF
 METHOD HAS-METHOD-NAME IS-METHOD-NAME-OF HAS-DOCUMENTATION IS-DOCUMENTATION-OF
...
 ALBERT BUTCHER BUTCHER-TRAINEE RED ZOE BOOK LIVRE HAS-TITRE HAS-OWNER IS-OWNER-OF
 SMITH MEDICAL-DOCTOR RESEARCHER BARTHES\;.LABROUSSE HAS-SISTER IS-SISTER-OF)
```

=get-instances (inverse-link)

=get-instances **Own method**

Gets all instances of concepts by extracting them from the $EILS property of `*moss-system*`.

Example:

```
? (send 'moss::$EIL '=get-instances)
(MOSS::$VALT.OF MOSS::$CNAM.OF MOSS::$XNB.OF MOSS::$TMBT.OF MOSS::$DOCT.OF
 MOSS::$ARG.OF MOSS::$REST.OF MOSS::$CODT.OF MOSS::$FNAM.OF MOSS::$MNAM.OF
 MOSS::$UNAM.OF MOSS::$SNAM.OF MOSS::$PRFX.OF MOSS::$SVL.OF MOSS::$CRET.OF
 MOSS::$DTCT.OF MOSS::$VERT.OF MOSS::$ENAM.OF MOSS::$LBLT.OF MOSS::$RDX.OF
 MOSS::$ONEOF.OF MOSS::$PNAM.OF MOSS::$MINT.OF MOSS::$MAXT.OF MOSS::$VRT.OF
 MOSS::$SEL.OF MOSS::$OPR.OF MOSS::$TPRT.OF MOSS::$NTPR.OF MOSS::$INAM.OF
 MOSS::$TYPE.OF MOSS::$DEFT.OF MOSS::$OBJNAM.OF MOSS::$TIT.OF MOSS::$DOCS.OF
 MOSS::$PT.OF MOSS::$PS.OF MOSS::$IS-A.OF MOSS::$OMS.OF MOSS::$IMS.OF MOSS::$CTRS.OF
 MOSS::$SYSS.OF MOSS::$REQS.OF MOSS::$EPLS.OF MOSS::$FNLS.OF MOSS::$ESLS.OF
 MOSS::$EILS.OF MOSS::$ETLS.OF MOSS::$ENLS.OF MOSS::$CLCS.OF MOSS::$INV.OF
 MOSS::$SUC.OF MOSS::$ONESUCOF.OF MOSS::$SUCV.OF MOSS::$NSUC.OF MOSS::$VNBT.OF
 MOSS::$CFNM.OF MOSS::$USRNAM.OF MOSS::$ACRT.OF MOSS::$CVRS.OF MOSS::$CFGS.OF
 MOSS::$USRS.OF MOSS::$REF.OF MOSS::$ID.OF MOSS::$QNAM.OF MOSS::$QXPT.OF
 MOSS::$QUXT.OF MOSS::$QANT.OF MOSS::$STNAM.OF MOSS::$XPLT.OF MOSS::$SOBT.OF
 MOSS::$DATT.OF MOSS::$DOMT.OF MOSS::$GOLT.OF MOSS::$QT.OF MOSS::$ACT.OF
 MOSS::$PRMT.OF MOSS::$MSGT.OF MOSS::$ANST.OF MOSS::$RSLT.OF MOSS::$NSTS.OF
 MOSS::$QRYT.OF MOSS::$WLT.OF MOSS::$ESTS.OF MOSS::$STS.OF MOSS::$ISTS.OF
 MOSS::$CTXS.OF MOSS::$DHDRS.OF MOSS::$OWS.OF MOSS::$IWS.OF MOSS::$TDO.OF
 MOSS::$AGT.OF MOSS::$FLT.OF MOSS::$ARNAM.OF MOSS::$ARKT.OF MOSS::$ARVT.OF
 MOSS::$ARQT.OF MOSS::$CREFT.OF MOSS::$ANAM.OF MOSS::$OPRT.OF MOSS::$ARGS.OF
 MOSS::$ACTS.OF MOSS::$GLS.OF MOSS::$T-MOSS-OBJECT-ARGUMENT-ARG-KEY.OF
 MOSS::$T-HOW-ACTION-OPERATOR.OF MOSS::$S-HOW-ACTION-OPERATOR-ARGUMENTS.OF
 MOSS::$T-PRINT-ACTION-OPERATOR.OF MOSS::$S-PRINT-ACTION-OPERATOR-ARGUMENTS.OF
 MOSS::$T-WHAT-ACTION-OPERATOR.OF MOSS::$S-WHAT-ACTION-OPERATOR-ARGUMENTS.OF
 $T-COLOR.OF $T-SHAPE.OF $T-DATE.OF $T-ORANGE-COLOR.OF $T-ORANGE-SHAPE.OF
 $T-FRUIT-COLOR.OF $T-TASTE.OF $T-FRUIT-TASTE.OF $T-APPLE-DATE.OF $T-BANANA-DATE.OF
 $T-NAME.OF $T-PERSON-NAME.OF $T-FIRST-NAME.OF $T-PERSON-FIRST-NAME.OF $S-FOOD.OF
 $S-PERSON-FOOD.OF $S-BROTHER.OF $S-PERSON-BROTHER.OF $T-SIZE.OF $T-TOWN-SIZE.OF
 $S-MOTHER.OF $S-PERSON-MOTHER.OF $T-SEX.OF $T-PERSON-SEX.OF $S-SISTER.OF
 $S-PERSON-SISTER.OF)
```

=get-instances (method)

=get-instances **Own method**

Get all instances of methods by extracting them from the $EILS property of `*moss-system*`.

Example:

```
? (send '$FN '=get-instances)
($FN.0 $FN.1 $FN.2 $FN.3 $FN.4 $UNI.0 $UNI.1 $UNI.2 $FN.5 $UNI.3 $UNI.4 $UNI.5
 $UNI.6 $UNI.7 $UNI.8 $UNI.9 $FN.6 $FN.7 $FN.8 $FN.9 $FN.10 $FN.11 $UNI.10 $UNI.11
 $UNI.12 $UNI.13 $FN.12 $FN.13 $FN.14 $FN.15 $UNI.14 $FN.16 $UNI.15 $UNI.16 $FN.17
 $FN.18 $FN.19 $FN.20 $FN.21 $FN.22 $FN.23 $FN.24 $UNI.17 $FN.25 $UNI.18 $UNI.19
 $UNI.20 $UNI.21 $UNI.22 $FN.26 $FN.27 $FN.28 $FN.29 $FN.30 $FN.31 $FN.32 $UNI.23
 $UNI.24 $UNI.25 $FN.33 $FN.34 $FN.35 $FN.36 $FN.37 $FN.38 $FN.39 $FN.40 $FN.41
 $FN.42 $FN.43 $UNI.26 $FN.44 $FN.45 $FN.46 $FN.47 $FN.48 $FN.49 $FN.50 $UNI.27
 $FN.51 $FN.52 $FN.53 $FN.54 $FN.55 $FN.56 $UNI.28 $FN.57 $FN.58 $FN.59 $FN.60
 $UNI.29 $UNI.30 $FN.61 $UNI.31 $UNI.32 $FN.62 $UNI.33 $UNI.34 $FN.63 $FN.64 $UNI.35
 $FN.65 $FN.66 $FN.67 $FN.68 $UNI.36 $UNI.37 $FN.69 $UNI.38 $UNI.39 $UNI.40 $UNI.41
 $FN.70 $FN.71 $FN.72 $FN.73 $FN.74 $FN.75 $FN.76 $FN.77 $UNI.42 $FN.78 $FN.79
 $UNI.43 $FN.80 $FN.81 $FN.82 $FN.83 $FN.84 $UNI.44 $FN.85 $FN.86 $FN.87 $FN.88
 $FN.89 $FN.90 $FN.91 $FN.92 $FN.93 $FN.94 $FN.95)
```

=get-instances (relation)

=get-instances **Own method**

Gets all instances of concepts by extracting them from the $ESLS property of `*moss-system*`.

Example:

```
? (send 'moss::$EPS '=get-instances)
(MOSS::$DOCS MOSS::$PT MOSS::$PS MOSS::$IS-A MOSS::$OMS MOSS::$IMS MOSS::$CTRS
 MOSS::$SYSS MOSS::$REQS MOSS::$EPLS MOSS::$FNLS MOSS::$ESLS MOSS::$EILS MOSS::$ETLS
 MOSS::$ENLS MOSS::$CLCS MOSS::$INV MOSS::$SUC MOSS::$ONESUCOF MOSS::$SUCV
 MOSS::$NSUC MOSS::$CFGS MOSS::$USRS MOSS::$NSTS MOSS::$ESTS MOSS::$STS MOSS::$ISTS
 MOSS::$CTXS MOSS::$DHDRS MOSS::$ARGS MOSS::$ACTS MOSS::$GLS
 MOSS::$S-HOW-ACTION-OPERATOR-ARGUMENTS MOSS::$S-PRINT-ACTION-OPERATOR-ARGUMENTS
 MOSS::$S-WHAT-ACTION-OPERATOR-ARGUMENTS $S-FOOD $S-PERSON-FOOD $S-BROTHER
 $S-PERSON-BROTHER $S-MOTHER $S-PERSON-MOTHER $S-SISTER $S-PERSON-SISTER)
```

=get-name (attribute)

=get-name **Instance method**

Returns the name of an instance of attribute.

Example:

```
? (send ‘$enam ’=get-name)
("CONCEPT-NAME")
? (send ’$T-PERSON-NAME ’=get-name)
("NAME")
```

=get-name (universal)

=get-name **Instance method**

Returns the name of an object using =instance-name. Same as =instance-name.

Example:

```
? (send '$ent '=get-name)
("CONCEPT")
? (send _person '=get-name)
("Person")
```

=get-properties **Universal method**

Gets list of all attributes and relations if a given entity from its model. If it is a classless object then returns the list in a random order.

Example:

- for a person

```
? (send _dp '=get-properties)
($T-NAME $T-FIRST-NAME $T-SEX $S-FOOD $S-BROTHER $S-MOTHER $S-SISTER)
```

- for a property

```
? (send '$T-NAME '=get-properties)
($INV $PNAM $MINT $MAXT $XNB $TMBT)
```

- for a concept

```
? (send '$FN '=get-properties)
($ENAM $RDX $XNB $TMBT $PT $PS $IS-A $OMS $IMS $CTRS $SYSS)
```

- for the meta-class

```
? (send 'moss::$ENT '=get-properties)
($ENAM $RDX $XNB $TMBT $PT $PS $IS-A $OMS $IMS $CTRS $SYSS)
```

- for an orphan

```
? (send _o2 '=get-properties)
($TYPE $T-SHAPE $T-COLOR)
```

- for an entry-point (yes, entry-points are objects!)

```
? (send '=get-properties '=get-properties)
($XNB $TMBT)
```

=get-user-classes (concept)

=get-user-classes **Own method**

Gets all instances of classes by extracting them from the $ENLS property of `*moss-system*`. All classes in the moss package are removed from the list.

Example:

```
? (send 'moss::$ent '=get-user-classes)
($E-ORANGE $E-FRUIT $E-APPLE $E-BANANA $E-PERSON $E-STUDENT $E-TOWN $E-TEACHER)
```

=get-user-concepts (concept)

=get-user-concepts **Own method**

Same as =get-user-classes.

=has-inverse-properties **Universal method**

When applied to any object returns the list of all inverse property ids. This is used in particular in connection with processing entry points.

Example:

```
? (send 'mary '=has-inverse-properties)
($T-NAME.OF $EPLT.OF)
```

=has-properties **Universal method**

When sent to an object, the method returns the list of all properties actually present in the object, with the exception of the property $TYPE, $ID and the inverse properties.

Example:

```
? (send _dp '=has-properties)
($T-PERSON-NAME $S-PERSON-FOOD MOSS::$OMS)
```

=has-value (universal)

=has-value *property-name* **Universal method**

Example: The following example shows the difference between =get and =has-value. In that case 25 is a default value which is not recognized by =has-value.

```
(defattribute AGE (:default 25)(:class person))
$T-PERSON-AGE
? (defindividual PERSON (HAS-NAME "Dumond")(:var _du))
$E-PERSON.9
? (send _du '=get'has-age)
(25) ; default value
? (send _du '=has-value 'has-age)
NIL
```

Error messages:

- An error occurs when the property does not exists:

```
? (send _du '=has-value 'has-money)
> Error: in =has-value HAS-MONEY is not a property of object ($E-PERSON.9) in context 0
> While executing: MOSS::*0=HAS-VALUE
> Type Command-. to abort.
See the Restarts? menu item for further choices.
1 >
```

Note: =has-value uses =has-value-id once the identity of the property has been established.

=has-value-id (universal)

=has-value-id *property-id* **Universal method**

Same as the previous one but uses internal id of property for faster response.

Example:

```
? (send du '=has-value-id _has-name)
("Dumond")
```

Note: The method uses the %get-value internal function in the current (default) context.

=id (entry point)

=id *property class* **Instance method**

Recovers the internal id of an object from its entry point, the associated property, and the class to which it belongs.

The result is either an object or a list of objects if more than one is found.

Examples:

```
? (send 'DUPOND '=id 'HAS-NAME 'PERSON)
$E-PERSON.1
? (m-definstance PERSON (HAS-NAME "Durand" "Dupond")(:var dd))
$E-PERSON.10
? (send 'DUPOND '=id 'HAS-NAME 'PERSON)
($E-PERSON.10 $E-PERSON.1)
```

Note: The method uses the %extract internal function in the current (default) context.

```
(%extract *self* property class :context *context* :allow-multiple-values T)
```

=if-added (attribute)

=if-added *value object-id* **Own method**

Demon for doing some book keeping after values have been added. Is normally called by =add-attribute-values-using-id.

The actual bookkeeping is left to the imagination of the user. It could be for example a redraw in a window.

Example:

```
(defownmethod =if-added _has-position (suc-id obj-id)
   "Redraws the diagram for showing the data"
(send *display-window* '=redraw-object obj-id value))
```

=if-added (relation)

=if-added *successor-id object-id* **Own method**

Demon for doing some book keeping after values have been added. Is normally called by =add-related-objects-using-id.

The actual bookkeeping is left to the imagination of the user. It could be for example a redraw in a window.

Example:

```
(defownmethod =if-added _has-brother (suc-id obj-id)
   "Redraws the diagram for showing the data"
  (send *display-window* '=redraw-link obj-id suc-id))
```

=if-removed (attribute)

=if-removed *value object-id* **Own method**

The method is a demon for doing some book keeping after values have been added. Is normally called by =delete.

The actual bookkeeping is left to the imagination of the user.

Example:

```
(defownmethod =if-removed _has-position suc-id obj-id)
   "Redraws the diagram for showing the change in data"
   (send *display-window* '=delete obj-id suc-id))
```

=if-removed (inverse link)

=if-removed *&optional old-entity-id old-suc-id entity-id suc-id* **Own method**

Daemon for doing book keeping after removing something. Default is to do nothing returning nil.

Optional arguments:

old-entity-id (opt): id of old entity

old-successor-id (opt): id of old successor

entity-id (opt): id of new (?) entity

successor-id (opt): id of new (?) successor

=if-removed (relation)

=if-removed *&optional old-entity-id old-suc-id entity-id suc-id* **Own method**

The method is a demon for doing some book keeping after links have been added. Is normally called by =delete.

The actual bookkeeping is left to the imagination of the user.

Optional arguments:

old-entity-id (opt): id of old entity

old-successor-id (opt): id of old successor

entity-id (opt): id of new (?) entity

successor-id (opt): id of new (?) successor

Example:

```
(defownmethod =if-removed _has-brother (suc-id obj-id)
   "Redraws the diagram for showing the change in data"
(send *display-window* '=undraw-link obj-id suc-id))
```

=increment (counter)

=increment **Instance method**

Increases value of counter by 1. Now does that in a special way since the counter must always increase regardless of the version in which it has been created.

=inherit-instance (universal)

=inherit-instance *method-name* **Universal method**

is the default inheritance mechanism implementing a lexicographic search (depth first) in case of multiple inheritance. The mechanism invoking this method is disabled in MOSS 7 rendering the method useless. However advanced users can restore the complex inheritance mechanism, which is however not detailed here.

=inherit-isa-instance (universal)

=inherit-isa-instance *method-name* **Universal method**

is the default inheritance mechanism when one tries to inherit an instance method starting with an orphan, and following prototype links to see whther one of the prototypes in the list is an instance of some class. The mechanism invoking this method is disabled in MOSS 7 rendering the method useless. However advanced users can restore the complex inheritance mechanism, which is however not detailed here

=inherit-own (universal)

=inherit-own *method-name* **Universal method**

is the default inheritance mechanism (lexicographic depth first) when one looks into the list of prototypes for a local method. The mechanism invoking this method is disabled in MOSS 7 rendering the method useless. However advanced users can restore the complex inheritance mechanism, which is however not detailed here.

=input-value (attribute)

=input-value *stream &rest l-text* **Instance method**

Examples: The last example shows that there is no connection between the =input-value method and the =format-value method which is applied afterwards.

```
? (send _has-name '=input-value)
HAS-NAME: Albert Jules Antoine
"Albert Jules Antoine"

? (send _has-name '=input-value "Valeur du nom:")
Valeur du nom: Zoe
" Zoe "

? (m-defownmethod =format-value _has-age (string-value)
    "age must be a number"
       (let ((age (read-from-string string-value)))
         (and (numberp age) age)))
AGE
$FN.81
? (send _has-age '=input-value)
HAS-AGE: 45
45
```

=instance-name **Instance method**

Returns the class name (as a list).

Example:

```
? (send _person '=instance-name)
("PERSON")
```

=instance-name **Instance method**

Returns the inverse name qualifying the inverse link.

Example:

```
? (send (send _has-brother '=inverse-id) '=instance-name)
("IS-BROTHER-OF")
```

=instance-name **Instance method**

Returns the name of the method that received the message.

Example:

```
? (send '$FN.35 '=instance-name)
=MAKE-ENTRY
```

=instance-name (property)

=instance-name **Instance method**

Returns the name of the property.

Example:

```
? (send _has-first-name '=instance-name)
("HAS-FIRST-NAME")
```

=instance-name (concept)

=instance-name *entity-id* **Instance method**

Returns a list summarizing entity to print - Default is first non nil terminal property - Maybe we should consider required properties ...

Example:

```
? (send 'moss::$ENT '=instance-summary _person)
("Person")
```

=instance-name (relation)

=instance-name *object-id* **Instance method**

Returns a summary of a relation, namely its name.

Example:

```
? (send _has-brother '=instance-summary _tom)
??
```

=inverse-id (attribute)

=inverse-id **Instance method**

Returns the id of the inverse attribute. Used for build entry points.

Example:

```
? (send _has-name '=inverse-id)
$T-NAME.OF
```

=inverse-id (inverse-link)

=inverse-id **Instance method**

Returns the property corresponding whose inverse link is the inverse.

The three methods for properties all use the same internal function: %inverse-property-id.*Example:*

```
? (send '$S-BROTHER.OF '=inverse-id)
$S-BROTHER
```

=inverse-id (relation)

=inverse-id **Instance method**

Returns the inverse internal id of the inverse relation. Needed to build inverse links.

Example:

```
? (send _has-brother '=inverse-id)
$S-BROTHER.OF
```

=kill-method (universal)

=kill-method *method-name* **Universal method**

Reaches all models following $IS-A.DE link - inverse of $IS-A - removing given method from prop-list - =kill-method must be used each time a method is suppressed to remove the code from the various p-lists onto which it was cached.

Warning: The method only acts on the p-list of the models that inherited the method. It cleans their cache. It does not act on instances nor does it delete the method. On standard use the p-list is deactivated (methods are not cached).

=kill-method is intended for advanced use.

=link (relation)

=link *entity-id successor-id* **Instance method**

Links two objects using the current relation. The link is semantically directed. Inverse links are only a facility for navigating in the reverse direction.

Example:

```
? (send _dd '=print-self)
----- $E-PERSON.10
 NAME: Durand,Dupond
 AGE: 25
-----
:DONE
? (send _mm '=print-self)
----- $E-PERSON.6
 NAME: Mary
 AGE: 25
-----
:DONE
? (defrelation WIFE  PERSON PERSON)
$S-PERSON-WIFE
? (send _has-wife '=link _dd _mm)
:DONE
? (send _dd '=print-self)
----- $E-PERSON.10
 NAME: Durand,Dupond
 AGE: 25
 WIFE: $E-PERSON.6
-----
:DONE
```

=make-entry (concept name)

=make-entry *data-string* **Own method**

is an own-method of the property CONCEPT-NAME. It is used to create entries for the names of new concepts.

It uses the internal function make-entry-point.

Example:

```
? (send '$ENAM '=make-entry "A very   sPecial property indeed")
(A-VERY-SPECIAL-PROPERTY-INDEED)
```

=make-entry (attribute name)

=make-entry *data-string* **Own method**

is an own-method of the property CONCEPT-NAME. It is used to create entries for the names of new classes.

It uses the internal function make-entry-point.

Example:

```
? (send 'moss::$pnam '=make-entry "albert est parti au boulot ; joseph l'Èpie")
(HAS-ALBERT-EST-PARTI-AU-BOULOT-\;-JOSEPH-L-EPIE)
```

=make-print-string (attribute)

=make-print-string *value-list &key header no-value-flag* **Instance method**

Produces a string with values associated to an attribute.

Arguments:

value-list: values associated to attribute

header (key): title to use instead of attribute name

no-value-flag (key): if t print even if value-list is nil

Example:

```
? (send _has-name '=make-print-string '("Albert" "GÈrard"))
"name: Albert, GÈrard"
```

=make-print-string (relation)

=make-print-string *suc-list &key header no-value-flag* **Instance method**

Produces a string with a summary of all linked entities.

Arguments:

suc-list: list of successors

header (key): title to use instead of relation name

no-value-flag (key): if t print even if value-list is nil

Example:

```
? (send _has-mother '=make-print-string '($E-PERSON.1 $E-PERSON.2))
("mother" "Barthes: Jean-Paul" "Barthes Biesel, Barthes, Biesel: Dominique")

? (send 'moss::$PT '=make-print-string '($T-PERSON-NAME $T-PERSON-FIRST-NAME))

("ATTRIBUTE " "name" "first-name")
```

=make-print-string (universal)

=make-print-string *&key (context *context*)* **Universal method**

Produces a list of strings for printing an entity by-passing model.

Arguments:

context for printing object in the particular context

Example:

```
? *language*
:FR
? (send 'albert::$E-person '=make-print-string)
("CONCEPT" ("CONCEPT-NAME" "personne") ("RADIX" ALBERT::$E-PERSON)
 ("ATTRIBUTE " "nom" "prÈnom")
 ("RELATION " "adresse domicile" "email" "page web" "epoux" "Èpouse" "mËre")
 ("COUNTER" ("COUNTER" ("VALUE" 1))))
```

=make-standard-entry (attribute)

=make-standard-entry *data-string* **Instance method**

Builds a list of symbols using the make-entry primitive. Spaces are removed and periods are inserted in between words. Argument may be a list or a string as shown in the examples.

Example:

```
? (send _has-name '=make-standard-entry ?The red shark?)
(THE-RED-SHARK)
```

This method is replaced whenever necessary by a user defined ownmethod attached to the corresponding attribute.

=modify-value (attribute)

=modify-value *entity-id value* **Instance method**

Modify current value - default is to ask for new value.

=new (attribute)

=new *&rest option-list* **Own method**

Defines the =new own-method for attributes for defining user models. Uses the same code as the m-make-attribute function with the following options:

```
(m-make-tp name &rest option-list)
```

Syntax of options is:

```
(:class <class-name>)   to attach to a class
(:class-id <class-id>)  same as above, but with internal
                        class id
(:default <value>) default, not obeying PDM specs
(:is-a <class-id>*) for defining inheritance
(:min <number>) minimal cardinality
(:max <number>) maximal cardinality
(:var <var-name>)       id.
(:unique)       minimal and maximal cardinality are
                        both 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)))
```

Done as follows:

- check if property already exists - if so, then error
- build property.

Example:

```
? (send 'moss::$EPT '=new 'nickname '(:max 2)
        '(:entry (value-list) "Entry for nickname"
                 (intern (moss::make-name value-list)))
        '(:class person))
$T-NICKNAME
```

=new (inverse link)

=new *prop-id* **Own method**

Creates a new instance of inverse-link e.g. $NAME.OF

Example:

```
? (send '$EIL '=new _has-name)
$T-NAME.OF
? (send '$EIL '=new '$T-PERSON-NAME)
$T-NAME.OF
```

=new (universal)

=new *&rest option-list* **Universal method**

Creates a new instance using the radix contained in the model e.g. $PERS.34. Uses =basic-new.

Example:

```
? (send _person '=new)
$E-PERSON.2
? $E-PERSON.2
(($TYPE (0 $E-PERSON)) ($ID (0 $E-PERSON.2)))
```

=new-classless-key (moss system)

=new-classless-key **Instance method**

Creates a new key for classless objects.

Example:

```
? (send moss::*moss-system* '=new-classless-key)
$0-5
```

=new-key *&rest option-list* **Universal method**

Deprecated.

=new-version (moss system)

=new-version *&rest option-list* **Own method**

Adding a new version to the system. Takes the last version of the version-graph adds 1, and forks from current version unless there is an option.

Options:

(`from` *old-branching-context*)
See the document about versions for a detailed explanation of the versioning mechanism.

(`from` *list-of-branching-contexts*)
See the document about versions for a detailed explanation of the versioning mechanism.

Example:

```
? (send *moss-system* '=new-version)
;***Warning: changing version and context from 0 to 1
1
```

Note: This method is intended for system use.

=normalize (attribute)

=normalize *value* **Instance method**

Normally normalizes data values - default is to do nothing.

=print (attribute)

=print *object-id* **Instance method**

Deprecated.

=print-all (universal)

=print-all *&key (stream *moss-output*)* **Universal method**

Prints an object bypassing its print function if present. It uses the method =get-properties to print the object content.

Uses the =print-value method for each property.

Example:

```
? (send _dp '=print-all)
----- $E-PERSON.1
 NAME: Dupond,Dupuis
 FIRST-NAME:
 SEX: unknown
 AGE: 25
 FOOD: $E-BANANA.1
 BROTHER:
 MOTHER:
 SISTER:
 WIFE:
-----
:DONE
 To be compared with:
? (send _dp '=print-self)

----- $E-PERSON.1
 NAME: Dupond,Dupuis
 SEX: unknown
 AGE: 25
 FOOD: $E-BANANA.1
-----
:DONE
or :
? (send _dp '=print-object)
----- $E-PERSON.1
 TYPE: $E-PERSON
 identifier: $E-PERSON.1
 NAME: Dupond,Dupuis
 FOOD: $E-BANANA.1
 OWN-METHOD: =HELLO OWN/ $E-PERSON.1, =HELLO-BIS OWN/ $E-PERSON.1,
     =HELLO-TER OWN/ $E-PERSON.1
-----
:DONE
or :
? (send _dp '=print-history)
$E-PERSON.1
----------
TYPE:                          t0: $E-PERSON
IDENTIFIER:                    t0: $E-PERSON.1
----- Attributes
NAME:                          t0: "Dupond", "Dupuis"
```

```
----- Relations
FOOD:                         t0: $E-BANANA.1
OWN-METHOD:                   t0: $FN.197, $FN.198, $FN.199
-----Inv-Links
----------
:DONE
```

=print-all-instances (concept)

=print-all-instances *&rest option-list* **Instance method**

Prints all instances of a class. The default is to print them all, using the =summary method which should have been defined (the default summary method prints the internal id of objects).

However one can print only a specific rance by using the `:range` option. One can also specify that we want the =print-instance method by using the `:full-print` option.

Options:

(`:range` *min max*)
Print instances in between two counter numbers. ‘

(`:full-print`)
Uses the method =print-self for printing each instance.

Examples:

```
? (send _person '=print-all-instances)
1 - $E-PERSON.1
2 - $E-PERSON.2
3 - $E-PERSON.3
4 - $E-PERSON.4
5 - $E-PERSON.5
6 - $E-PERSON.6
7 - $E-PERSON.7
"*done*"

? (definstmethod =summary PERSON () "returns first-name names"
     (append (or (HAS-FIRST-NAME)(list "unknown first name"))
             (or (HAS-NAME)(list "unknown name"))))
$FN.83

? (send _person '=print-all-instances)

1 - "unknown first name" "Dupond" "Dupuis"
; Warning: object with id: $E-PERSON.2 has been erased in this context (0)when trying
 to apply method:
;             =SUMMARY, with arg-list: NIL
; While executing: SEND-NO-TRACE

2 - "unknown first name" "Dupond" "Dupuis"
3 - "unknown first name" "Tom"
4 - "unknown first name" "George"
5 - "unknown first name" "Tommy"
6 - "unknown first name" "unknown name"
7 - "unknown first name" "Tommy"
:DONE"

? (send _person '=print-all-instances '(:range 3 5) '(:full-print))

 NAME: Tom
 SEX: unknown
```

```
 AGE: 25
-----
 NAME: George
 SEX: unknown
 AGE: 25
-----
 NAME: Tommy
 SEX: unknown
 AGE: 25
-----
:DONE
```

=print-as-method-for (entry point)

=print-as-method-for **Instance method**

Looks at an entry point for a possible name of a method. If so, for each object prints the instance method and local method corresponding to the name if any.

Example:

```
? (send '=get-name '=print-as-method-for)

(=GET-NAME)
Returns the name of an instance $ENAM or $PNAM
Arguments: NIL
Function:
 ((SEND-NO-TRACE *SELF* '=INSTANCE-NAME))

----- $UNI.25
 UNIVERSAL-METHOD-NAME: =GET-NAME
 DOCUMENTATION: Returns the name of an instance $ENAM or $PNAM
 CODE: ((SEND-NO-TRACE *SELF* '=INSTANCE-NAME))
 FUNCTION-NAME: *0=GET-NAME
-----
:DONE
```

=print-code (method)

=print-code *&key (stream *moss-output*)* **Instance method**

Prints Function code in pretty format.

Key argument:

stream (key): printing stream default t

Example:

```
? (send '$FN.39 '=print-code)
("Return a list summarizing entity to print - Default is first non nil terminal property
 - Maybe we should consider required properties ...
Arguments:
   entity-id: id of entity to print
Return:
   A list summarizing entity"
 (IF (%IS-MODEL? ENTITY-ID)
     (SEND ENTITY-ID '=INSTANCE-NAME)
     (LET ((PROP-LIST (G==> 'HAS-TERMINAL-PROPERTY)))
       (WHILE (AND PROP-LIST (NOT (SEND ENTITY-ID '=GET-ID (POP PROP-LIST)))))
       *ANSWER*)))
:DONE
```

=print-doc (method)

=print-doc *&key (stream *moss-output*)* **Instance method**

Prints Function name and associated documentation.

Key argument:

stream (key): printing stream default t

Example:

```
? (send '$FN.39 '=print-doc)
Return a list summarizing entity to print - Default is first non nil terminal property
- Maybe we should consider required properties ...
Arguments:
   entity-id: id of entity to print
Return:
   A list summarizing entity
:DONE
```

=print-documentation (universal)

=print-documentation *&key (stream *moss-window*) no-summary (lead "") final-new-line erase &allow-other-keys* **Universal method**

Prints object documentation or sorry message.

Key argument:

stream (key): printing stream default t

erase (key): if t and stream is a pane, then erases the pane

no-summary (key): if t does not print a summary of the object

final-new-line (key): if t prints a final new line

lead (key): if t prints a leading header

Example:

```
? (send _person '=print-documentation)

PERSON : Model of a physical person
:DONE
```

=print-error **Universal method**

Deprecated.

=print-history (universal)

=print-history **Universal method**

Print the content of an object. For each property prints all values in all contexts from the root of the version graph to the current context.

Uses the `%pep` internal function.

Example:

```
$E-PERSON
----------
TYPE:                         t0: MOSS::$ENT
IDENTIFIER:                   t0: $E-PERSON
----- Attributes
CONCEPT-NAME:                 t0: (:EN "PERSON")
RADIX:                        t0: $E-PERSON
DOCUMENTATION:                t0: (:EN "Model of a physical person")
----- Relations
COUNTER:                      t0: $E-PERSON.CTR
ATTRIBUTE:                    t0: $T-PERSON-NAME, $T-PERSON-FIRST-NAME,
                                  $T-PERSON-NICK-NAME, $T-PERSON-AGE,
                                  $T-PERSON-BIRTH-YEAR, $T-PERSON-SEX
RELATION:                     t0: $S-PERSON-BROTHER, $S-PERSON-SISTER,
                                  $S-PERSON-HUSBAND, $S-PERSON-WIFE,
                                  $S-PERSON-MOTHER, $S-PERSON-FATHER,
                                  $S-PERSON-SON, $S-PERSON-DAUGHTER,
                                  $S-PERSON-NEPHEW, $S-PERSON-NIECE,
                                  $S-PERSON-UNCLE, $S-PERSON-AUNT,
                                  $S-PERSON-GRAND-FATHER,
                                  $S-PERSON-GRAND-MOTHER,
                                  $S-PERSON-GRAND-CHILD, $S-PERSON-COUSIN
INSTANCE-METHOD:              t0: $FN.198
-----Inv-Links
IS-ENTITY-LIST-OF:            t0: MOSS::$SYS.1
IS-SUCCESSOR-OF:              t0: $S-PERSON-BROTHER, $S-PERSON-SISTER,
                                  $S-PERSON-HUSBAND, $S-PERSON-WIFE,
                                  $S-PERSON-MOTHER, $S-PERSON-FATHER,
                                  $S-PERSON-SON, $S-PERSON-DAUGHTER,
                                  $S-PERSON-NEPHEW, $S-PERSON-NIECE,
                                  $S-PERSON-UNCLE, $S-PERSON-AUNT,
                                  $S-PERSON-GRAND-FATHER,
                                  $S-PERSON-GRAND-MOTHER,
                                  $S-PERSON-GRAND-CHILD, $S-PERSON-COUSIN,
                                  $S-ORGANIZATION-EMPLOYEE
IS-IS-A-OF:                   t0: $E-STUDENT
----------
:DONE
```

t0 indicates context 0.

=print-instance (concept)

=print-instance *instance-id &key (stream *moss-output*)* **instance method**

Print an instance using model to get list of properties.

Example:

```
? (send _person '=print-instance  _dp)

 NAME: Dupond,Dupuis
 SEX: unknown
 AGE: 25
 FOOD: $E-BANANA.1
-----
:DONE
```

=print-local-methods (universal)

=print-local-methods *&key (stream *moss-output*)* **Universal method**

Prints all methods and documentation associated with the current object.

Example:

```
? (send _has-name '=print-local-methods)


INHERITED OWN METHODS
=====================

=MAKE-ENTRY
Entry-point function for HAS-NAME

INSTANCE METHODS OBTAINED FROM CLASS
====================================

=ADD
Add a value or a list of new values. Duplicates are NOT discarded. Values
 are normalized using the =xi method, then added at the end of the current list.
 The =if-added method is checked. Whenever it fails, the corresponding
 successor is not added. Cardinality constraints are checked for maximal
 value. If too many values are specified, then some of them are discarded.
 Minimal cardinality is not checked and a warning is not issued.
Arguments:
   obj-id: ID of object to modify
   value-list: list of data
   option-list (opt):
       (:before data) to insert list in front of the specific data which must be
 in internal normalized format
       (:before-nth nth) to insert list in the nth position (useful when there are
 several identical values - that can happen with TPs).
Return:
   the internal representation of the modified object.

=CHECK
Check that the value list attached to the attibute in object obeys the various r
estrictions attached to the attribute.
   Prints eventual errors using mformat.
Arguments:
   value-list: list of object identifiers (presumably attached to the attribute)
Return:
   value-list if OK, nil otherwise.

=DELETE
Delete a value for a given object. The value is normalized using the =xi method
 if any has been defined. Eventual entry points are removed.
Arguments:
   entity-id: id of object
```

```
   value: value to delete
   option-list (opt):
        (:nth position) the position of the value is given in case there are several
identical values for the attribute
Return:
   internal representation of object.

=DELETE-ALL
Delete current values associated with the property of object. The result is a null
value associated with the property.
Arguments:
   entity-id: id of object
Return:
   internal representation of object.

=FORMAT-VALUE
Format single value associated to property. A property can use such a
function for a special formatting.
   when value is a multilingual name, we extract national canonical part otherwise
we return the value unchanged.
Argument:
   value: value to format
Return:
   value reformatted if MLN, value otherwise.

=FORMAT-VALUE-LIST
Format a list of values, presumably attached to a property. We limit the
 length of the string to 80 characters, to avoid problems with Pascal strings.
Arg1: value-list

=GET-NAME
Returns the name of an instance $ENAM or $PNAM

=IF-ADDED
Daemon for doing book keeping after adding something. Default is to do
nothing returning value. If error is detected then returning nil aborts data.
Arguments:
   value: new value
   entity-id: id of object (in case it is needed)
Return:
   value if OK, nil otherwise.

=IF-REMOVED
Daemon for doing book keeping after removing something Default is to do
nothing returning nil.
Arguments:
   value: value that was removed
   entity-id: object-id
Return:
   nil
```

```
=INPUT-VALUE
Input value associated with a given terminal property.
Arguments:
   stream: output channel or window pane
   l-text: optional header for property (default property name
Return:
   normed value from a read.

=INSTANCE-NAME
returns attribute name with associated class name if class keyword is true
Arguments:
   class (key): t or nil
Return:
   list of name and class name eventually, e.g. ("name") or ("name/person")

=INVERSE-ID
Returns the internal id of the inverse attribute. No args.

=MAKE-STANDARD-ENTRY
Makes standard entry points using make-entry. Returns a list of symbols that will
be used as ids for the entry points.
Arguments:
   data: used to build the entry point (may be a list)
Return:
   a list of symbols

=MODIFY-VALUE
Modify current value - default is to ask for new value.
Arguments:
   entity-id: id of entity whose property must be modified
   value: value to be modified

=NORMALIZE
Normally normalizes data values - default is to do nothing.
Arguments:
   value: value to normalize
Return:
   normalized value

=PRINT
Print a terminal-property - Arg1: entity-id

=PRINT-VALUE
Print values associated to property - A property can uses =print-value for a
special formatting.
Arguments:
   value-list: values associated to attribute
   stream (key): stream or pane (default t)
   header (key): title to use instead of attribute name
```

```
   no-value-flag (key): if t print even if value-list is nil
Returns:
   nil

=SUMMARY
Return the property name

=XI
Normally normalizes data values. Checks value restriction conditions associated
 with the terminal property.
Arguments:
   data: list of values to be checked.
Return:
   data if OK, nil otherwise
:DONE
```

=print-methods (universal)

=print-methods *&key (stream *moss-output*)* **Universal method**

Prints all methods and documentation associated with instances of the current object which should be a class. Methods redefined locally, i.e. at a particular instance level, are ignored.

Example:

```
? (send _person '=print-methods)


LOCAL INSTANCE METHODS
======================

=SUMMARY
Extract first names and names from a person object
:DONE
```

=print-name (concept)

=print-name *&key (stream *moss-output*)* **Instance method**

Prints onto the current active output the name of the class.

Example:

```
? (send _person '=print-name)
Person
"Person "
```

=print-object (universal)

=print-object *&key no-inverse (stream *moss-output*))* **Universal method**

Print the content of an object by sending a =print-value message to all its properties including inverse properties except when the option specifies not to print inverse properties.

Options:

(`:no-inverse`)
Instruction not to print inverse properties.

(`:stream` *stream*)
Allows redirecting printing towards a specific stream

Example:

```
? (send _dp '=print-object)
----- $E-PERSON.1
 TYPE: $E-PERSON
 identifier: $E-PERSON.1
 NAME: Dupond,Dupuis
 FOOD: $E-BANANA.1
 OWN-METHOD: =HELLO OWN/ unknown first name Dupond Dupuis,
     =HELLO-BIS OWN/ unknown first name Dupond Dupuis,
     =HELLO-TER OWN/ unknown first name Dupond Dupuis
-----
:DONE
```

=print-self (entry point)

=print-self *&key (stream *moss-output*))* **Instance method**

Instead of printing the entry point itself, actually prints all objects associated with a given entry point.

Example: Compare the result of using =print-self vs using =print-object:

```
? (send 'dupond '=print-self)

----- $E-PERSON.1
 NAME: Dupond,Dupuis
 SEX: unknown
 AGE: 25
 FOOD: $E-BANANA.1
-----
----- $E-STUDENT.3
 NAME: Dubois,Dupond
-----
----- $E-PERSON.9
 SEX: unknown
 AGE: 25
 FOOD: $E-BANANA.1
-----
:DONE

? (send 'dupond '=print-object)
----- DUPOND
 TYPE: $EP
 identifier: DUPOND
 IS-NAME-OF: unknown first name Dupond Dupuis, unknown first name Dubois Dupond,
     unknown first name unknown name
 IS-ENTRY-POINT-LIST-OF: MOSS
-----
:DONE
```

=print-self (method)

=print-self *&key (stream *moss-output*))* **Instance method**

Prints the content of the method: name, documentation, code, etc.

Example:

```
? (send '$FN.63 '=print-self)
(=PRINT-SELF)
Print objects corresponding to an entry point.
Arguments
   stream (key): output stream or pane (default t)
Return:
   :done
Arguments: (&KEY (STREAM T))
Function:
 ((LET ((PROP-LIST (DELETE-DUPLICATES (SEND *SELF* '=HAS-INVERSE-PROPERTIES))))
   (WHILE PROP-LIST
          (WHEN (AND (%IS-INVERSE-PROPERTY? (CAR PROP-LIST))
                     (NOT (EQL (CAR PROP-LIST) (%INVERSE-PROPERTY-ID '$EPLS))))
            (BROADCAST (%GET-VALUE *SELF* (CAR PROP-LIST))
                       '=PRINT-SELF
                       :STREAM
                       STREAM))
          (POP PROP-LIST))
   :DONE))
:DONE
```

=print-self (universal)

=print-self *&key (context *context*) (stream *moss-output*))* **Universal method**

Prints any object in a nicer format that =print-object.
Works for any object in a default fashion, unless the method was redefined.

Options:

(`:context` *nn*)
Printing context for the object.

(`:stream` *stream*)
Allows redirecting printing towards a specific stream.

Example:

- instance of person

```
? (send _dp '=print-self)

----- $E-PERSON.1
 NAME: Dupond,Dupuis
 SEX: unknown
 AGE: 25
 FOOD: $E-BANANA.1
-----
:DONE
```

- orphan

```
? (send _o2 '=print-self)

----- $0-1
 TYPE: *NONE*
 identifier: $0-1
 SHAPE: Square
 COLOR: Red
-----
:DONE
```

- class

```
? (send _person '=print-self)

----- $E-PERSON
 CONCEPT-NAME: PERSON
 RADIX: $E-PERSON
 DOCUMENTATION: Model of a physical person
 ATTRIBUTE : NAME/PERSON, FIRST-NAME/PERSON, NICK-NAME/PERSON, AGE/PERSON,
     BIRTH YEAR/PERSON, SEX/PERSON
 RELATION : BROTHER/PERSON, SISTER/PERSON, HUSBAND/PERSON, WIFE/PERSON,
     MOTHER/PERSON, FATHER/PERSON, SON/PERSON, DAUGHTER/PERSON, NEPHEW/PERSON,
     NIECE/PERSON, UNCLE/PERSON, AUNT/PERSON, GRAND-FATHER/PERSON,
```

```
    GRAND-MOTHER/PERSON, GRAND-CHILD/PERSON, COUSIN/PERSON
 INSTANCE-METHOD: =SUMMARY INST/ PERSON
 COUNTER: 22
-----
:DONE

? (send _student '=print-self)

----- $E-STUDENT
 CONCEPT-NAME: STUDENT
 RADIX: $E-STUDENT
 DOCUMENTATION: A STUDENT is a person usually enrolled in higher education
 ATTRIBUTE : NAME/PERSON, FIRST-NAME/PERSON, NICK-NAME/PERSON, AGE/PERSON,
     BIRTH YEAR/PERSON, SEX/PERSON
 RELATION : COURSES/STUDENT, BROTHER/PERSON, SISTER/PERSON, HUSBAND/PERSON,
     WIFE/PERSON, MOTHER/PERSON, FATHER/PERSON, SON/PERSON, DAUGHTER/PERSON,
     NEPHEW/PERSON, NIECE/PERSON, UNCLE/PERSON, AUNT/PERSON,
     GRAND-FATHER/PERSON, GRAND-MOTHER/PERSON, GRAND-CHILD/PERSON,
     COUSIN/PERSON
 IS-A: PERSON
 INSTANCE-METHOD: =SUMMARY INST/ PERSON
 COUNTER: 7
-----
:DONE
```

- methods

  Methods are objects but have their own =print-self method.

```
? (send 'moss::$FN.22 '=print-self)

----- =FILTER
Can be defined so as to filter successor of a given entity.

   Default action is Don't filter

Arguments:

   suc-id: id of successor

   entity-id: id of entity

Return:

   suc-id if filtering predicate is satisfied, nil otherwise.
Function:
 NIL
:DONE
```

One can see on this last example the properties inherited from the class PERSON.

=print-typical-instance (concept)

=print-typical-instance *&key (stream *moss-output*))* **Instance method**

Prints a typical instance of a class, corresponding to its ideal, i.e., the instance bearing all the default attributes.

Example:

```
? (send _person '=print-typical-instance)

----- $E-PERSON.0
 SEX: unknown
 AGE: 25
-----
:DONE
```

=print-value (attribute)

=print-value *value-list &key (stream *moss-output*) header no-value-flag)* **Instance method**

Printing function attached to attributes. Value-list is the set of values to be printed.

Options:

(`:stream` *stream*)
if present specifies the output stream.
(`:header` *header*)
if present specifies text to be printed instead of the property name.
(`:no-value-flag` *t/nil*)
if true prints property even if it has no value.

Example:

```
? (send _has-name '=print-value (HAS-NAME _dp))

 NAME: Dupond, Dupuis
NIL

? (send _has-name '=print-value (HAS-NAME _dp) :header "Nom")

 Nom: Dupond, Dupuis
```

=print-value (inverse link)

=print-value *successor-list &rest option-list* **Instance method**

Prints a summary of all linked objects.
Mainly used internally by the system.

Example:

```
? (send '$S-BROTHER.OF '=print-value (has-brother '$E-PERSON.17))

 IS-BROTHER-OF: unknown first name Dubois Dupond
NIL
```

=print-value (relation)

=print-value *successor-list &rest option-list* **Instance method**

Printing function attached to structural properties. Successor-list is the set of successors to be printed. The description printed for each successor is a list obtained by sending the message =summary to each of them in turn.

Return the string corresponding to what was printed.

Options:

(`:header` *header*)
if present, text to be printed instead of the property name,

Example:

```
? (defrelation wife person person)
$S-WIFE
? (defindividual PERSON (HAS-NAME "Barthes" "Biesel")(HAS-FIRST-NAME "Dominique")
   (:var _dbb))
$E-PERSON.8
? (defindividual PERSON (HAS-NAME "Barthes")(HAS-FIRST-NAME "Jean-Paul")(HAS-WIFE dbb)
   (:var _jpb))
$E-PERSON.9

? (send _has-wife '=print-value (HAS-WIFE _jpb))

 WIFE: Dominique Barthes Biesel
NIL
? (send _has-wife '=print-value (HAS-WIFE _jpb) :header "Epouse ")

 Epouse : Dominique Barthes Biesel
NIL
```

=print-warning (universal)

=print-warning *message* **Universal method**

Deprecated.

=replace (universal)

=replace *property-ref value-list* **Universal method**

Replaces the value list associated with a given property with the specified value list. Use the methods =delete-all and =add associated with the specific property.

Example:

```
? (send _jpb '=print-self)

----- $E-PERSON.21
 NAME: Barthes
 FIRST-NAME: Jean-Paul
 SEX: unknown
 AGE: 25
 WIFE: Dominique Barthes Biesel
-----
:DONE
? (send _jpb '=replace "age" '(61))
((MOSS::$TYPE (0 $E-PERSON)) (MOSS::$ID (0 $E-PERSON.21))
 ($T-PERSON-NAME (0 "Barthes")) ($T-PERSON-FIRST-NAME (0 "Jean-Paul"))
 ($S-PERSON-WIFE (0 $E-PERSON.19)) ($T-PERSON-AGE (0 61)))
? (send _jpb '=print-self)

----- $E-PERSON.21
 NAME: Barthes
 FIRST-NAME: Jean-Paul
 SEX: unknown
 AGE: 61
 WIFE: Dominique Barthes Biesel
-----
:DONE
```

Note: =replace uses =delete-all and =add methods associated with the specific property.

Error message:

```
? (send _jpb '=replace 'HAS-MONEY '("Hector"))
"No known HAS-MONEY Property for object $E-PERSON.21"
```

=save-application (moss-system)

=save-application *&optional (application-name *application-name*)* **Universal method**

Save all the objects of an application including system objects into a file whose name is <application name>.mos

System objects are saved last.

Warning: This method is still experimental. It saves a copy of the core image of the application. However, this disconnects the application from the text files containing the definitions. It can be considered as a cheap way to implement persistency.

Example:

Messages:

```
;*** saving the world into #P"Odin:Users:barthes:MCL:MOSS:MOSS
v6.0.0aM:applications:test.mos"reference to $0-0 which does not exist in context 0
,Warning: unbound object: $UNI.55
,Warning: unbound object: $UNI.51
,Warning: unbound object: $UNI.50
```

Most warnings correspond to ghost objects that are created by the function making instance keys from the class counter. This is especially true when the class is shared by the system and by the user (e.g. methods).

=summary (attribute)

=summary **Instance method**

Returns the property name.

Example:

```
? (send _has-age '=summary)
("Age/Person")
```

=summary **Instance method**

Normally specified by the user; by default returns a list with the model name.

Example:

- classes

```
? (send _person '=summary)
("Person")
```

- meta-classes

```
? (send 'moss::$ENT '=summary)
("CONCEPT")
```

=summary **Instance method**

Returns the value of the counter.

Example:

```
? (send 'moss::$CTR '=new)
$CTR.2
? (send * '=summary)
(0)
```

=summary (inverse link)

=summary **Instance method**

Returns the inverse property name.

Example:

```
? (send '$S-BROTHER.OF '=summary)
((:EN "IS-BROTHER-OF"))
```

=summary (method)

=summary **Instance method**

Returns the name of the method and its type and associated class.

Example:

```
? (send 'moss::$FN.22 '=summary)
(=FILTER MOSS::INST/ "RELATION")
```

=summary **Instance method**

Returns the property name.

Example:

```
? (send _has-brother '=summary)
("BROTHER/UNIVERSAL-CLASS")
```

=summary (moss system)

=summary **Instance method**

Example:

```
? (send moss::*moss-system* '=summary)
(MOSS)
```

=summary (universal)

=summary **Universal method**

is a default method for returning a list to be used as a summary of any object. The default summary is the internal id of the object!

The method is there to be specialized for each class of objects. It is in particular called by printing functions when they have to list successors of a given structural property.

Example:

```
? (defrelation course person course)
$S-COURSE

? (send ' $E-STUDENT.1 '=add-related-objects "course" _ia1)
((moss::$TYPE (0 $E-STUDENT)) ($T-NAME (0)) ($S-BROTHER.OF (0 $E-PERSON.1))
 ($S-COURSE (0 $E-COURSE.1)))

? (send '$E-STUDENT.1 '=print-self)

----- $E-STUDENT.1
 COURSE: $E-COURSE.1
-----
:DONE

? (definstmethod =summary COURSE () "retourne le code UV" (HAS-CODE-UV))
$FN.85

? (send '$E-STUDENT.1 '=print-self)

----- $E-STUDENT.1
 COURSE: IA01
-----
:DONE
```

This example shows how the =summary method is used when printing the content of a student object. Without the method, MOSS prints the identifier of the course ($E-COURSE.1). When the method is defined, then MOSS prints the code of the course.

=summary (universal method)

=summary **instance method**

Return the name of the universal method.

Example:

```
? (send 'moss::$UNI.22 '=summary)
(=DELETE-SP-ID)
```

=unknown-method **Universal method**

is the default method for taking care of unknown methods. The default is to print a message stating that the method is not available for the specific object and continuing execution.

Example:

```
? (definstmethod =unknown-method COURS (method-name)
    "Special way of handling errors by throwing to the :unknown tag"
   (throw :unknown (format nil "Unknown method: ~A" method-name)))
$FN.199

? (catch :unknown (send '$E-COURSE.1 '=zorch))
"Unknown method: =ZORCH"
```

=unlink (inverse link)

=unlink *object-id successor-id* **Instance method**

Removes a link between the two objects (same as =delete).

=unlink (relation)

=unlink *object-id successor-id* **Instance method**

Removes the links between two objects.

=what? (universal)

=what? **Universal method**

Gives a summary of the type of object we are considering, including documentation if available, and the list of ancestors of the model.

Example:

```
? (send _person '=what?)
The object is an instance of ENTITY
*Sorry, no specific documentation available*
T

? _dd
$E-STUDENT.1

? (send dd '=what?)
The object is an instance of STUDENT
... has ancestors (depth first)
PERSON
*Sorry, no specific documentation available*
T
```

=xi (attribute)

=xi *data* **Instance method**

Normally normalizes data values. Checks value restriction conditions associated with the attribute. Uses the internal %validate-tp-value to check each possible restriction that could apply to the value. Returns the value if OK, NIL otherwise.

It is the same method as =normalize.

Chapter 5

Dialogs

Contents

5.1 Introduction

MOSS provides a general mechanism for developing natural language dialogs in the favorite language of the user (e.g. English, French, Japanese, etc.). 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 (5.1).

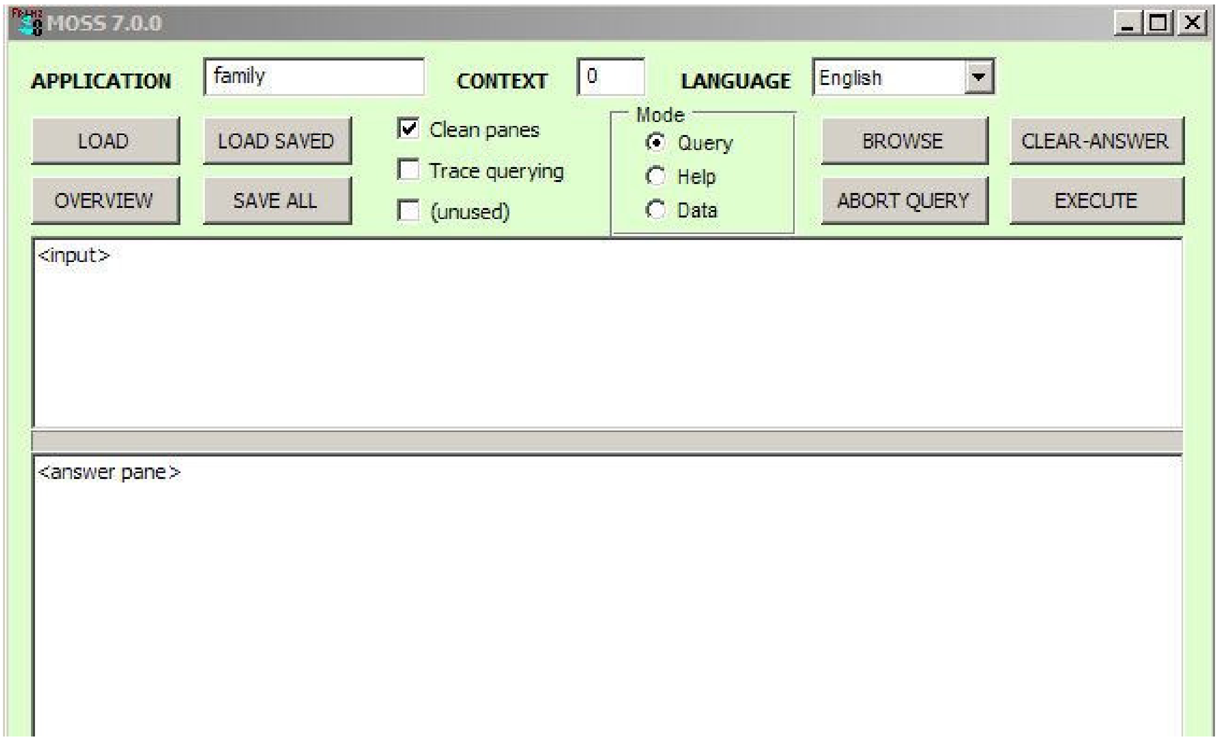

Figure 5.1: MOSS interaction window

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.

5.2 Example of Dialog

The interaction between the user and MOSS occurs normally through the MOSS window (5.2) using the Help mode. Under certain conditions it can be carried out in the Lisp listener, which is easier for obtaining a transcript.

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
```

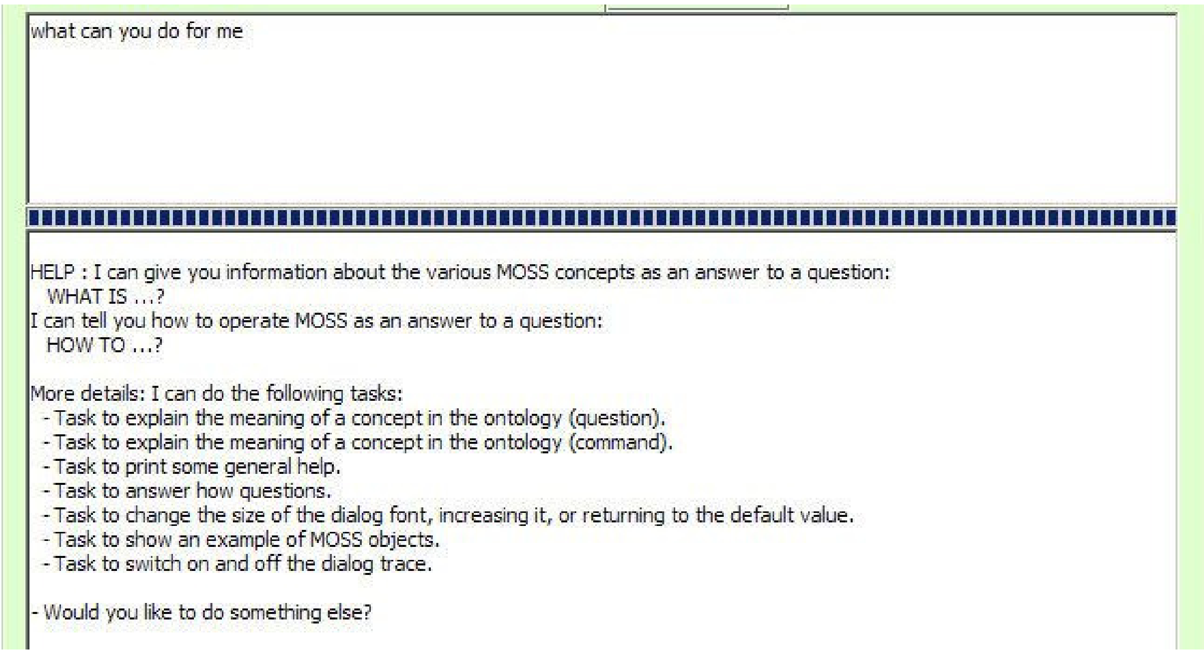

Figure 5.2: MOSS dialog in the interaction window (Help mode)

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 BarthËs 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.

5.3 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.

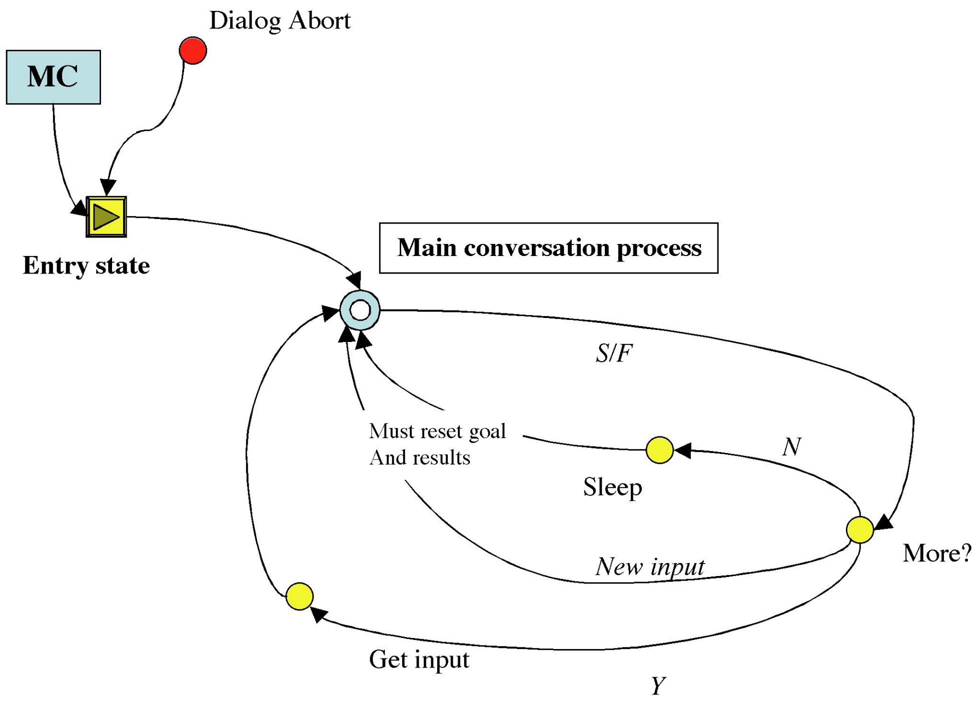

Figure 5.3: Example of conversation graph (OMAS Personal Assistant)

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

5.3.1 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.

5.3.2 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.

5.3.3 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.

=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 and computes a transition.

5.4 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 naÔve 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 naÔve user can use it intuitively.

5.4.1 Example

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

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.

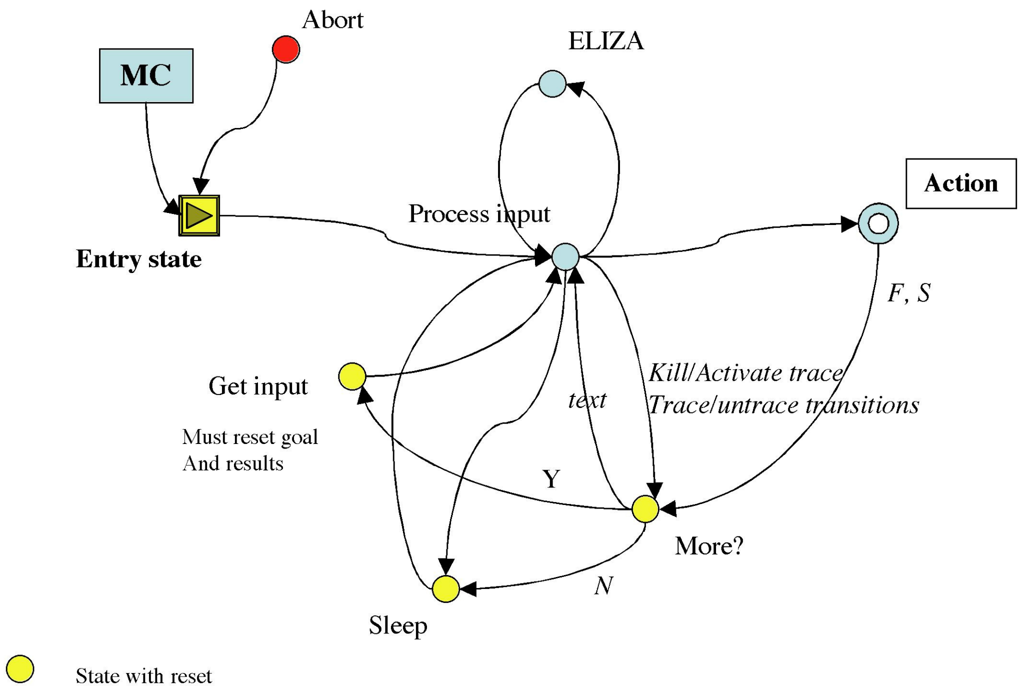

Figure 5.4: MOSS Main conversation graph

Building an Initial Graph

From the set of questions, now try to imagine a possible conversation graph (Fig.5.4).

Fig.5.4 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").

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.

  ```
  (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 ([**?**]). 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."

  As can be seen Fig.5.5 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)
```

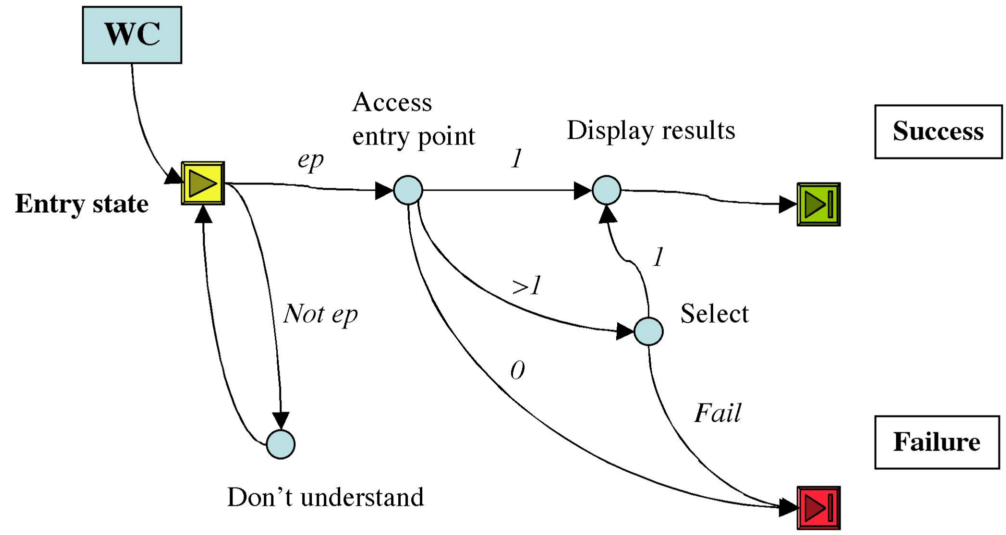

Figure 5.5: What is conversation (subdialog)

```
 (: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 print
  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, sim
 (: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.5.6. It can be seen that subdialogs are called corresponding to the type of action to comment upon: create, query, kill, delete, modify.

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.

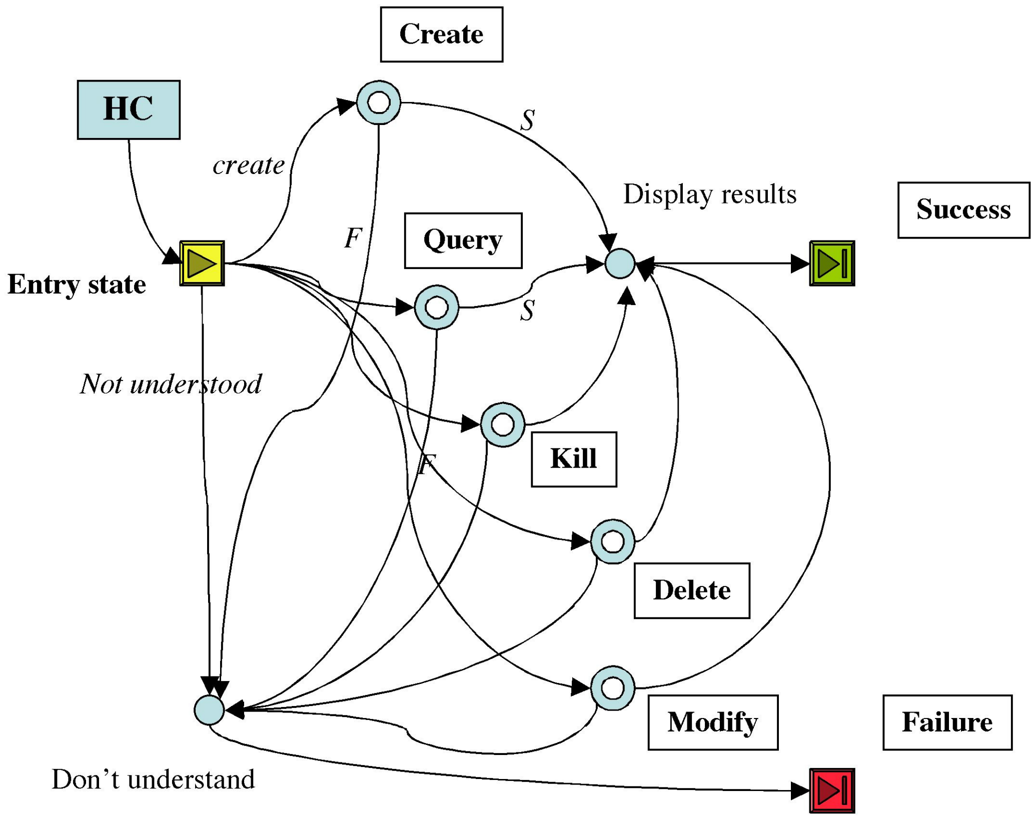

Figure 5.6: Graph of the "How" subdialog

5.5 Dialog Process

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

5.5.1 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.

5.5.2 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.

5.5.3 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.

5.5.4 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.

5.5.5 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.

5.6 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.

5.6.1 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
-----
```

5.6.2 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.
-----
```

5.6.3 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
-----
```

5.6.4 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: DÈbut du dialogue
 INITIAL-STATE: DÈbut du dialogue
 DIALOG-HEADER: $QHDE.13
 SUB-DIALOG-HEADER: $QHDE.14
 Q-CONTEXT:
     ALBERT::$SCXTE.967 label: NIL
     DATA : NIL
     RESULTS:
     WORKING-LIST: NIL
-----
```

5.6.5 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.

5.7 Dialog Language

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

5.7.1 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.

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.

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

5.7.2 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)
   ))
```

5.7.3 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.

5.7.4 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))
```

Index

Chapter 6

Multilinguism

Contents

6.1 Introduction

When developing the SOL representation for building the Terregov ontology, we had a requirement to accommodate several languages, namely, French, English, Italian and Polish. Because MOSS had been developed in the context of a single language (English), it has to be modified to support other languages.

Thus a new datatype was introduced, the multilingual name abbreviated as mln. The idea was to be able to use a single piece of data for representing the same text in different languages. At the same time a global variable *language* would specify which language to choose from. In addition we felt the need to represent synonyms or equivalent expressions in a given language, a possibility that we added to the representation.

A library of elementary functions was developed and used to integrate the new feature to MOSS.

The resulting possibilities have been used in the Terregov project but also in the OMAS multi-agent platforms, where we can have Personal Assistant agents speaking different languages, although using the services of the same multilingual Service agents.

6.2 Multilingual Names

A multilingual name has been defined as a list containing language tags, formally:

```
<mln> ::=  ({:name} {<language tag> <terms>}+)
<terms> ::= "<term> {; <term>}"
```

where curly brackets indicate that the term is optional and + indicates one or more occurrence of the expression and term represents any combination of words.

Example

The following expressions are multilingual names:

```
(:name :en "Person")
(:name :en "Person; Fellow")
(:name :en "Person; Fellow" :fr "Personne; Quidam")
(:en "Person" :fr "Personne" :it "Persona")
(:fr "Personne; Quidam")
(:en "Alice in Wonderland" :fr "Alice au pays des merveilles")
```

Person and Fellow are here considered as English synonyms, Personne and Quidam as French synonyms.

6.3 Function Library

Several functions have been created for dealing with multilingual names (Table 6.1) and synonyms (Table 6.2).

Table 6.1: MLN functions

%mln?(expr)	type check
%mln-add-value(mln value language-tag)	add a language tag and value
%mln-equal(mln1 mln2 &key language)	test for equality
%mln-extract (mln &key (language *language*) canonical)	extract language string
%mln-extract-all-synonyms(mln)	gets all synonyms
%mln-extract-mln-from-ref (ref language &key no-throw-on-error (default :en))	builds a language mln
%mln-filter-language (mln language-tag &key always)	extract synonym string
%mln-get-canonical-name (mln)	extract a canonical name
%mln-get-first-name (name-string)	extract first synonym from a string
%mln-member (input-string tag mln)	check for membership
%mln-merge (&rest mln)	merge several mlns
%mln-norm (expr)	remove the :name tag
%mln-print-string (mln &optional (language-tag :all))	print an mln nicely
%mln-remove-language (mln language-tag)	remove language entry
%mln-remove-value (mln value language-tag)	remove a value
%mln-set-value (mln language-tag syn-string)	set language tag and value
%multilingual-name? (expr)	same as %mln?

Table 6.2: Synonym functions

%synonym-add(syn-string value)	adding a synonym
%synonym-explode (text)	extract all synonyms
%synonym-make (&rest item-list)	makes a synonym string
%synonym-member (value text)	membership test
%synonym-merge-strings (&rest names)	merge synonyms
%synonym-remove (value text)	removes a synonym

6.4 Using Multilingual Names

Using multilingual names is sometimes tricky. Some ontologies do not have entries for all languages for a number of reasons. Another difficult point is for creating or using objects in a multilingual context. This is detailed in the next two sections.

6.4.1 Inputting Multilingual Data

Two cases occur when inputting data into MOSS: (i) all data use the same language; of (ii) data are in different language. In the latter case two sub-cases can occur: (i) data are tagged with the corresponding language: or (ii) data are mixed and not tagged.

Inputting in the Same Language

Whenever the data uses the same language, it is not necessary to define mlns in the definitional macros of MOSS, although it is correct.

For example: writing

```
(defconcept (:name :en "Person")
    (:att (:en "name; family name") (:entry))
    (:att (:en "first name"))
    )
```

or writing

```
(defconcept "Person"
    (:att "name; family name" (:entry))
    (:att "first name")
    )
```

is equivalent, provided that in the second case the global `*language*` variable be set to `:EN`.

Note that the value of the global `*language*` variable will shadow any other language specified in the definition.

Inputting in Different Languages

If we want to specify several languages in the input data, then the global `*language*` variable should be set to the value `:ALL`. In that case we must tag the different strings. E.g.:

```
(defconcept (:name :en "Person" :fr "Personne")
    (:att (:en "name; family name") (:entry))
    (:att (:en "first name" :fr "prÈnom"))
    )
```

Note that the system will accept the data even if some tags are missing like in the example where the name attribute has no French value. However, it is an error to give simple strings.

The :unknown tag: There are some cases however when the data is multilingual but cannot be tagged because it comes from an external source. If we want to include the new data in a multilingual environment, then we must add a tag to the language. The special tag `:unknown` can be declared as the value of the global `*language*` variable.

6.4.2 Accessing Data

Accessing data is done with regard to the value of the global `*language*` variable.

The solution we adopted is the following: if a name is requested in a specific language, then we try to obtain it using the specific language tag. However, if it is not present, then we try to get the English name for it. If the latter is not present, we try to find an `:unknown` tag. If we cannot find it we take a value choosing randomly from the languages that are recorded.

6.4.3 Legal Languages

As mentioned previously languages are specified by the use of (standard) tags. The list of languages recognized by MOSS is given by the value of the global `*language-tags*` variable.

6.5 Miscellaneous

Several points should be considered when using multilingual data:

- Most tools cannot accommodate languages other than those that can be expressed in ASCII.
- It is recommended to encode all strings using UNICODE (e.g. UTF-8). Now, if this option is selected, UTF-8 encoded files should be loaded using the `external-formal` option.
- One should be careful when specifying the input language when creating MOSS concepts.

Chapter 7

Multilinguism

Contents

7.1 Introduction

The paper contains a description of some advanced features of the MOSS 7.xx programming interface. It is intended for the application designers who are concerned with efficiency, or who simply want to have a deeper insight on some of the issues. Some internal functions can be called directly instead of using kernel methods, which avoids going through the process of sending messages. When doing so, of course one is more efficient, however one looses the message tracing facilities.

The reader is referred to Chapter **??** for a description of the underlying model. In the following text we use the terms class and concept interchangeably. Likewise instance and individual.

7.2 Synonyms, Multilingual Names and References

7.2.1 Class Names, Property Names and Synonyms

Concept and function names are now allowed to be strings or multilingual names rather than symbols.

```
(defconcept HOUSE (:rel ADDRESS ADDRESS))
```

can be now written

```
(defconcept ?House? (:rel ?address? ?Address?))
```

The case for letters in the string is not important, however, it will be used for printing.

The consequences are the following:

- the name of the concept and properties does not depend on the package in which the functions are executed.
- it is now possible to add synonyms by separating them with a semi-column.

```
(defconcept ?House; building?
    (:rel ?address; living place? ?Address?))
```

7.2.2 Multilingual Names

Furthermore, it is possible to add multilingual names in the following way:

```
(defconcept (:name :en ?House; building? :fr ?maison?)
   (:rel (:name :en ?address; living place? :fr ?adresse?)
         ?Address?))
```

The leading marker `:name` is optional.

Thus, most of the names can be replaced by strings or multilingual names and are now called references. Most functions have been rewritten to accept references instead of symbols denoting classes or properties.

Note however that the string referring to properties does not contain the prefix ?HAS-?, MOSS being able to make the difference between classes and properties bearing the same name from the position of the reference.

7.3 Entry Points

Entry points are actually keys computed by the system to be used as indexes to access corresponding objects.

There is a default method =make-entry used when the `:entry` option is specified in one of the definition macros. When applied to a value (list) the method returns an atom, capitalized, each separate word being separated by a dash, e.g.

```
"Robin Hood"  yields the atom ROBIN-HOOD
```

Otherwise, a user-defined function can be specified with the :entry option, in which case the result must be a list of atoms, built in any way suitable to the user. For example the INTERMARC format for coding the French titles of books in catalog applications removes the leading article, then keeps the first 3 letters of the first word and then the first letter of the following three words, e.g.

```
"Le Petit Chaperon Rouge"  yields PETCR
```

Of course when one uses special functions for formatting entry points, one must apply them in exactly the same fashion to produce the key to access the corresponding objects.

Entry points need not be unique. Furthermore, entry points are instances (individuals) of an entry point concept. They usually do not have direct properties but rather contain a list of pointers to objects grouped by inverse attributes.

Since properties are multi-valued, an entry point is built for each value associated with the attribute. Nothing in practice prevents the =make-entry method to build several entry points associated with the same value. This can be interesting for range indexing, although the feature was never really tested.

7.4 Recovering an Internal Object Id

Internal object-ids are meaningless to the user. In addition, they can change during a prototyping phase when objects are created by downloading a text definition file. Thus, they cannot be used directly. However all messages must specify object-ids. When object-ids are contained in variables one uses the variable. When no variable was attached to the object (e.g. through the `:var` option in the defxx macros), then the object-ids must be retrieved from the system. The =id method attached to the class of entry points does the job.

=id *attribute class* **method**

=id returns the internal obj-id of an object when sent to the entry point corresponding to an attribute. The result is a list of all objects corresponding to the entry point.

Example

```
(send 'JONES '=id "name" "person")
```

returns the list of all persons (obj-ids) with name JONES.

This is not a query mechanism as can be found in the MOSS/QS subsystem.

See also the service function `%extract` for faster access.

It is also possible to use a query (access function) to recover the list of objects, e.g.

```
(access 'JONES)
```

or, using more control,

```
(access '("person" ("name" :is "Jones")))
```

7.5 Selectors and Filtering Functions

<must be rewritten>

7.6 Generic Properties

A MOSS property may be created in different ways: (i) directly, i.e. independently from any class; or (ii) within the context of a class (while defining a class).

In each case, a generic property is created first unless it already exists. Then, if the property is a class property, a local property is created, attached to the class.

Example:

```
? (defattribute "weight" (:entry) (:max 1))
$T-WEIGHT
```

The property weight is created as a generic property independently from any class. It can be used by any object in the system. However, a new property with the same name can be created and at the same time be attached to a class, e.g.

Example:

```
? (defattribute "weight" (:entry) (:class "person"))
$T-PERSON-WEIGHT
```

In that case the new property is created as sub-property of the generic property.

When creating a property attached to a class, the class must already exist, unless it is being defined, and a generic property is first created, then a specific sub-property attached to the class is being created. The name of the two properties are the same.

Example:

```
? (defconcept "person" (:att "name" (:entry) (:min 1)))
$E-PERSON
? (defindividual "person" ("name" "John")(:var _john))
$E-PERSON.2
? _john
$E-PERSON.2
? (send _john '=get "name")
("John")
? $E-PERSON.2
(($TYPE (0 $E-PERSON)) ($ID (0 $E-PERSON.2)) ($T-PERSON-NAME (0 "John")))
```

Warning: there is no check between the logical consistency of the options of a generic and a local property. I.e. a generic property could specify a maximal cardinality of 2, and a specific property with the same name a minimal cardinality of 3. The condition on the local property will shadow that of the global property. This point should probably be revised in the future. Example:

```
? (defattribute "color" (:max 1))
$T-COLOR
? (defattribute "color" (:class "person")(:min 2))
$T-PERSON-COLOR
? (send '$t-color '=get "maximal-cardinality")
(1)
? (send '$t-person-color '=get "minimal-cardinality")
(2)
```

When the generic property is first created, it takes the restrictions that are defined at that time. When creating local properties later with the same name, any restriction will apply only locally. I.e., the restrictions of the generic property will not be changed.

7.7 Multi-Class Belonging

It is now possible to create individuals belonging to several classes.

Example:

```
(defconcept "teacher" (:is-a "person"))
$E-TEACHER
? (defconcept "student" (:is-a "person"))
$E-STUDENT
? (defindividual ("student" "teacher") ("name" "albert"))
$E-STUDENT.1
? (send '$e-student '=get-instances)
($E-STUDENT.1)
? (send '$e-teacher '=get-instances)
($E-TEACHER.1)
? (send '$e-student.1 '=get "name")
("albert")
? (send '$e-teacher.1 '=get "name")
("albert")
```

Albert is a Teaching Assistant and a doctoral student, and thus can be considered as a teacher or as a student.

Internally there is a single object representing Albert. The second object is a reference to the first one (same technique as the Lisp "invisible pointer".

Warning: if we use the access function to access persons, then Albert will appear twice, once as a teacher and another time as a student:

```
? (access '("person"))
($E-PERSON.1 $E-PERSON.2 $E-PERSON.3 $E-TEACHER.1 $E-STUDENT.1)
```

This should be fixed in the future.

7.8 Internal MPS Functions

The message passing system is constituted by a set of internal functions that can be called directly. The functions rely on a format that implements objects as association lists. The MPS functions implement message passing and method inheritance. If they are modified, it is quite possible for the system to collapse.

7.8.1 Message Passing Primitive Functions

There are actually 4 basic functions: send, send-no-trace, broadcast, send-super.

send *object-id method-name &rest arguments* **function**

object-id must evaluate to the internal id of an object (otherwise see the remark below) *method-name* to the name of a method, *arguments* are the optional arguments required by the method.

Example:

```
(send _p1 '=get 'HAS-NAME)
(send _p1 '=add-tp-id '$T-PERSON-NAME "Albert")
```

assuming that _p1 contains the obj-id of the person we want to send the message to (e.g. $E-PERSON.34), and that $T-PERSON-NAME represents the obj-id of the property NAME.

The send function takes care of the tracing machinery and then calls another internal function send-no-trace. The latter function is necessary since the tracing mechanism, implemented by sending message, would enter an infinite loop if we had only one send function.

send-no-trace *object-id method-name &rest arguments* **function**

Mainly used by the system for avoiding either entering an infinite loop, or cluttering the screen with low level trace messages when tracing all traffic.

It is not advised to use this function, because when it is used the corresponding messages cannot be printed by the tracing mechanism. The overhead incurred by using the send function when tracing is off is not very significant.

broadcast *object-id-list method-name &rest arguments* **function**

The function takes the same arguments as send with the exception of the first one that must evaluate to a list of object-ids. The broadcast is not a parallel operation but more a serial broadcast to each object in turn (like a let* for example). It returns the list of all answers returned by each object to the message.

Example:

```
(broadcast (list _p1 _p2) '=get "first name")
returns
(("Jean") NIL)
```

sends to both _p1 and _p2 a message requesting the list of their first names. A nil value could come from an object with unknown first name, or no first name attribute.

send-super *method-name arg-list* **function**

The function is analogous to the SMALLTALK send-super, and is used within a given method code to acquire the method with the same name defined at a higher level.

This function should be used carefully.

7.8.2 Method Inheritance and Associated Functions

See the next paragraph on "The Inheritance Mechanism."

7.9 The Inheritance Mechanism

Inheritance uses two possible mechanisms:

1. a complex mechanisms allowing to modify inheritance at each stage by using inheritance methods,
2. a simplified traditional lexicographic mechanism.

The first mechanism is described in Sections 7.9.1 through 9.3. The second one is described in Section 9.4. The default mechanism is the simple one.

7.9.1 General Mechanism

When a given object receives a message it must locate the corresponding code. This is done using the following algorithm:

1. if the method name refers to an own method locally available, then the local own method is used;
2. if the own method is not available locally, then it is inherited from the ancestors of the receiving object. To do so MOSS sends to the receiving object a new message =inherit-own with argument the needed method name:
   (a) if a method =inherit-own exists locally then it is used to look for the method originally wanted;

   (b) otherwise the process recurses until eventually the default universal method =inherit-own is used, in which case inheritance is done lexicographically (depth first).

3. when no own method can be found, then one tries to obtain an instance method from the class of the object (classless objects are a special case). When the method is not readily available, then an inheritance mechanism similar to the previous one is used replacing =inherit-own with =inherit-instance.

4. When everything fails, one looks for a universal method. If there exists one, then it is used;

For classless objects, at step 2 one follows the own-method inheritance path until an object instance of a class is encountered. As soon as this happens the request is passed to the corresponding class for an instance method.

7.9.2 Method Inheritance

The inheritance mechanism is implemented by the get-method function. It calls three sub-functions, which have a very similar behavior as can be seen in the following descriptions of the algorithms we use.

moss::get-method *object-id method-name* **function**

The function returns the method name associated with the object whose id is passed as the second argument. It does so according to the inheritance procedure explained previously.

moss::get–instance-method *object-id method-name* **function**

The detailed algorithm is as follows:

1. If the object-id is nil, then we quit.

2. If the object already has the instance method we are looking for, cached locally, then we use it. If it has the indication that no such method is available, then we return nil.

3. If the method is available at the site as an instance-method of the object, then we reconstruct the LISP function from the method object building an internal unique function name, and we cache the result locally.

4. If everything failed so far and we have no more parents, then we record that the method is not available for this object, and we return nil.

   Here, we did not find the method, but we have some parents. We must explore the parents in the same order as the =inherit-instance has done, with the particularity that we are looking for an instance method and not for an own-method. Hence we are going to send the message =inherit-instance to the object (*self*). However, if the method we are looking for is =inherit-instance itself, then we must be careful no to enter an infinite loop.

5. If the method we are looking for is =inherit-instance, then since the default strategy for inheritance is depth first, we organize a depth first strategy to locate a possible special =inherit-instance. Hence we try to locate the special method depth first among the ancestors. When this fails the function returns nil.

6. If the method is not =inherit-instance, then we send ourselves the message =inherit-instance with the name of the method we are looking for. If we find it, then we cache it.

7. If everything failed, then we record that an instance method is not available for such an object.

moss::get–own-method *object-id method-name* **function**

The detailed algorithm is as follows:

1. If the object-id is nil, then we quit.
2. If the object already has the instance method we are looking for, cached locally, then we use it. If it has the indication that no such method is available, then we return nil.
3. If the method is available at the site as an own-method of the object, then we reconstruct the LISP function from the method object building an internal unique function name, and we cache the result locally.
4. If everything failed so far and we have no more parents, then we record that the method is not available for this object, and we return nil.

   Here, we did not find the method, but we have some parents, then we must explore the parents. Hence we are going to send the message =inherit-own-instance to the object (*self*). However, if the method we are looking for is =inherit-own-instance itself, then we must be careful not to enter an infinite loop.
5. If the method we are looking for is =inherit-own-method, then since the default strategy for inheritance is depth first, we organize a depth first strategy to locate a possible special =inherit-own-method. Hence we try to locate the special method depth first among the ancestors. When this fails the function returns nil.
6. If the method is not =inherit-own-method, then we send ourselves the message =inherit-own-method with the name of the method we are looking for. If we find it, then we cache it.
7. If everything failed, then we record that an own method is not available for the object.

moss::get-isa-instance-method *object-id method-name* **function**

The function is called when trying to inherit an own-method along the IS-A link did not work. The idea is thus to follow the is-a link until one find an object instance of a class, in which case one tries to inherit an instance method from the class as usual. If this does not work, then one follows the IS-A link one step further and tries again, until we end up with some object having no parents, in which case we quit. When coming down the exploratory tries, we mark the passed classes with either the method if we found one, or with the indication that there is none if we found none.

It is important that we go up the IS-A inheritance link with the =inherit-own method to follow the inheritance path in the same fashion as when we looked for an own-method.

The detailed algorithm is as follows:

1. If the object-id is nil, then we quit.
2. If the object already has the instance method we are looking for, cached locally, then we use it. If it has the indication that no such method is available, then we return nil.
3. If the object is an instance of some class, then we call the get-instance-method on its class. Note that we do not allow an object to belong to multiple classes, otherwise this part should be changed.

4. If everything failed so far and we have no more parents, then we record that the method is not available for this object, and we return nil.

   Here, we did not find the method, but we have some parents. We must explore the parents in the same order as the =inherit-own has done, with the particularity that we are looking for an instance method and not for an own-method. Hence we are going to send the message =inherit-isa-instance to the object (*self*). However, if the method we are looking for is =inherit-isa-instance itself, then we must be careful no to enter an infinite loop.

5. If the method we are looking for is =inherit-isa-instance, then since the default strategy for inheritance is depth first, we organize a depth first strategy to locate a possible special =inherit-isa-instance. Hence we try to locate the special method depth first among the ancestors. When this fails the function returns nil.

6. If the method is not =inherit-isa-instance, then we send ourselves the message =inherit-isa-instance with the name of the method we are looking for. If we find it, then we cache it.

7. If everything failed, then we record that an instance method is not available for such an object.

The trick is to have an =inherit-isa-instance method that works exactly as the =inherit-own.

moss::get-universal-method *object-id method-name* **function**

The function is called when the other inheritance functions failed. MOSS then tries to find a universal method corresponding to the method name. If there exists one, then it returns it otherwise it return a null method.

moss::find-own-method *object-id method-name* **function**

The function tries to locate a method locally, i.e. within the list of methods attached to the object. If none is found then returns nil.

moss::find-instance-method *object-id method-name* **function**

The function tries to locate a method locally, i.e. within the list of methods attached to the object. If none is found then returns nil.

7.9.3 Local Service Functions

moss::parents *object-id* **function**

The function returns the list of all objects parents of the current object. If none, then it returns an empty list.

moss::ancestors *object-id* **function**

Same as parents but returns the transitive closure of the parent objects.

moss:: gamma *object-id* **function**

The function is used to compute a transitive closure along any of the structural properties.

7.9.4 Simplified Inheritance

The fancy inheritance mechanism described in the above paragraphs is not always very useful. Thus, in simple situations, the user can turn it off in favor of a more traditional inheritance scheme, i.e. a lexicographic one (inheritance depth first).

The inheritance mode is controlled through the moss::*lexicographic-inheritance* global variable. When MOSS is booted in a standard fashion the inheritance mode is the simple lexicographic one.

7.10 Methods Caching

7.10.1 Rationale

Some level of optimization was introduced to avoid inheritance look up at each cycle. Methods are cached at the class level for instance methods and at the object level for own methods onto the p-list of the corresponding obj-ids. Failures are also recorded.

When methods are cached, then execution is slower at the beginning of a work session, but much faster after that.

7.10.2 The moss::*cache-methods* Global Variable

The caching mechanism is difficult to use when developing a new application, because methods have to be modified constantly. Hence a special global variable *cache-methods* can disable the process of caching them locally. This leads to a slower execution, since methods are inherited again and again for the same objects, but it is much safer and prevents obscure errors.

By default caching is disabled. To turn it on, it is enough to set the global variable moss::*cache-methods* to T.

7.11 Internal Service Functions

Many functions are provided for increased efficiency. They deal explicitly with versions and are rather difficult to use. The reader should refer to the corresponding Manual: MOSS 7: Low Level Functions. The following functions are given as examples.

moss::%extract *att class &key context filter class-ref function allow-multiple-values* **function**

Option (:filter <function>) When the filter option is specified then function is applied to the whole list of objects prior to returning them. If the result returns more than one value and the option allow-multiple-values is not set to T, an error is signaled.

Example

```
(moss::%extract 'SMITH 'HAS-NAME 'PERSON :context 25 :allow-multiple-values t)
```

will try to extract the persons, students,..., named Smith in context (version) 25.

moss::%extract-from-id *entry att-id class-id &optional context* **function**

Returns the list of all objects corresponding to entry-point *entry*, to the attribute *att-id* and to the class *class-Id*, in the specific *context.* Specifying the context is optional; default is current context.

Objects are returned either if they are instances of the class or of one of the subclasses, regardless of the system to which classes belong (i.e. a sub-class may belong to a different system than its superclass).

The function is in fact a first a first attempt to locate objects, that will be further filtered by the %extract function.

Example

```
(moss::%extract-from-id 'ATTRIBUTE 'moss::$ENAM 'moss::$ENT 0)
```

returns

```
(MOSS::$EPT)
```

moss::%make-ep *entry tp-id obj-id &optional context* **function**

Add a new object to an entry point if it exists otherwise create it and record the creation in the MOSS system.

Returns the value of the entry point (useful when using step).

Example

```
(moss::%make-ep 'DE.LA.PORTE '$T-PERSON-NAME '$E-PERSON.356)
```

7.12 The Error Mechanism

In case of a failure (no method corresponding to the message could be located) MOSS sends the special message =unknown-method to the receiving object. The whole process starts again.

7.13 Trace and Stepping Mechanisms

7.13.1 Global Control Variables

moss::*trace-flag* controls the printing of the messages sent and of the values that are returned.

moss::*trace-level* sets the column indentation for starting printing the trace.

7.13.2 Trace functions

Tracing can be done in different ways: global tracing of all exchanged messages, tracing of a particular method, or monitoring the message traffic for a particular object.

trace-message **function**
allows monitoring message traffic between objects.

untrace-message **function**
reverts to no trace situation.

untrace-all **function**
applies to all trace modes.

trace-method *method-name* **function**
prints messages using a particular method.

untrace-method *method-name* **function**
suppresses monitoring messages for the method method-name.

trace-object *object-id* **function**
prints the message traffic related to a particular object.

untrace-object *object-id* **function**
reverts to no monitoring.

7.14 The Meta-Model Level (Kernel)

Fig.7.1 pictures the bottom up approach and Fig.7.2 the abstraction/specification relationships between the instances and the classes (models or structural specifications).

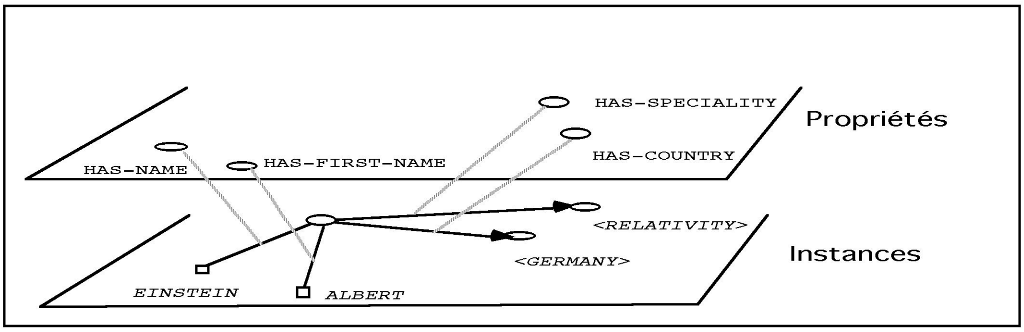

Figure 7.1: A class (concept) can be defined by abstracting the structure of the objects HAS-TP indicates an attribute (immediate value) HAS-SP indicates a relation (link onto another object)

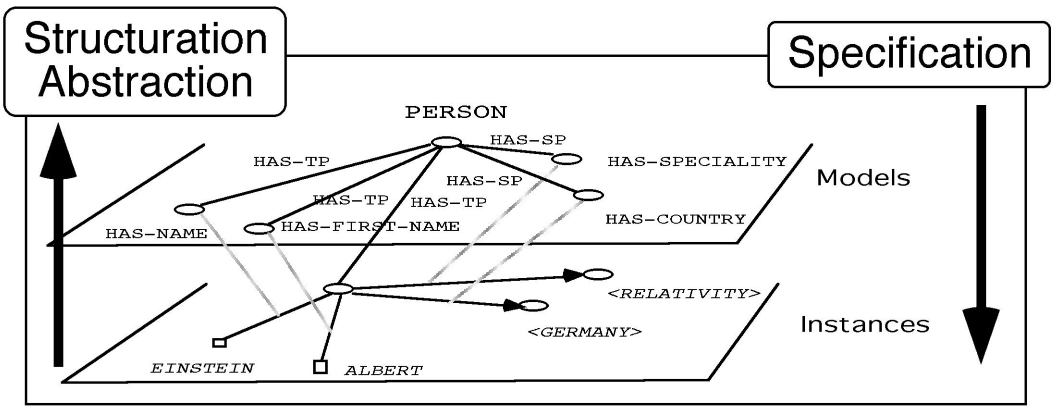

Figure 7.2: Objects (entities) are defined as soon as properties (quiddities) are defined. Properties specify the object structure.

Fig.7.3 shows for the same model the (multiple) inheritance structure. Inheritance is a class/subclass relationship among objects. At instance level such links become prototyping links, in particular for orphans. Such classless objects are interesting for design applications or for recognition (like in robotics).

The main point here is a clear distinction between specification and prototyping, which was already mentioned by Stein 87.

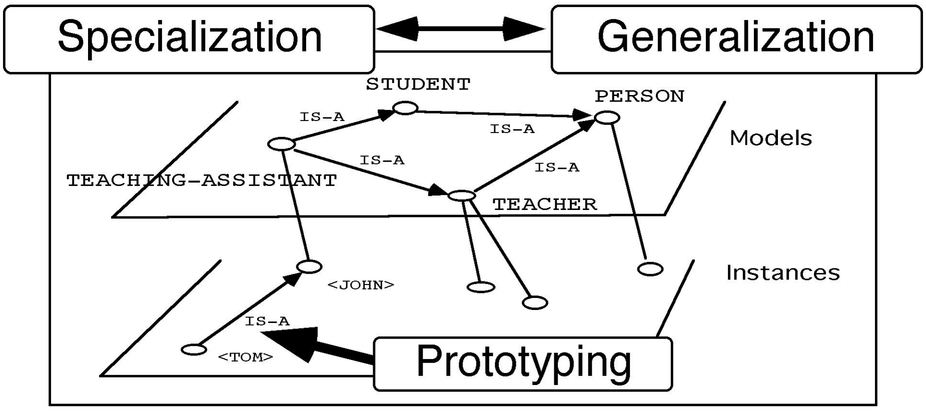

Figure 7.3: Specialization/generalization relationships are represented in the upper layer - In the instance layer the same is-a link is used to implement a prototyping relationship in particular for orphans.

Once classes have been defined, the abstraction process can be repeated at their level. Classes may be abstracted, i.e. their structure can be specified by meta-classes. In turn meta-classes can be abstracted, and so on until one finds an object which specifies its own structure (named here meta-meta-model). Fig.7.4 displays a 4-level hierarchy showing the reflectivity of the PDM model. Reflectivity is a powerful concept, since it allows to modify dynamically the structure of the model itself; however it is difficult to handle (see also Maes 87). Reflectivity can be the possibility of describing its own structuration like here. In KSL (Ibrahim & Cummins 88) the system specifies also the programming syntax of the language.

Reflectivity is important because it allows modeling the structure of the used representation. This is good for automatically providing **explanations**, and also for **learning**.

The two higher levels of the model must be hand-crafted, they constitute the kernel of the representation model.

7.15 Semantics of the Inverse Links

Inverse links do not bear any semantics. They are purely internal links to allow navigation along the arcs in both directions. This is used by the query system to optimize accesses. It is used of course by the browser. Thus no integrity constraint are attached to inverse links, e.g. there cannot be things like a minimal or maximal cardinality. The facility of giving specific names to inverse links is intended to make rephrasing of queries more legible.

It is important to notice that if a person is linked to another person by the property HAS-FATHER, then the inverse link is by default IS-FATHER-OF, but it does not appear in the printing list of the second object. If one wants to state that the second object is linked to the first by a property HAS-CHILD, then one must do it explicitly. This may look somewhat heavy, but the benefits are quite substantial in the query system. I tried once to give a meaning to inverse properties, thus leading to integrity constraints attached to the inverse links. This leads to severe locking problems in the concurrency control mechanism.

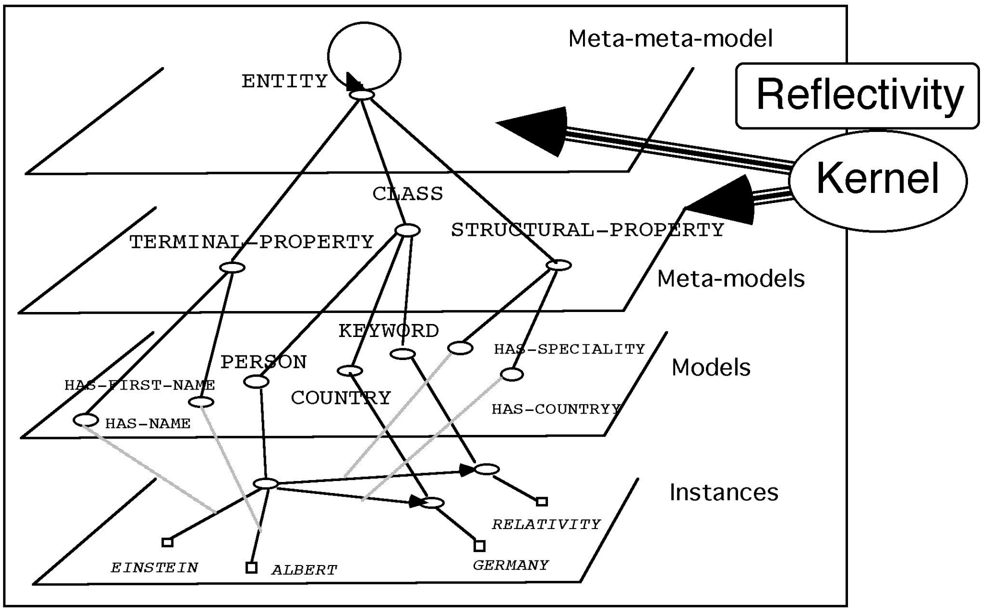

Figure 7.4: A (very simplified) representation of the various layers of the PDM4 model

7.16 Default Values

Semantic representations are expected to accommodate default values. A natural way of storing them is at the property level. However, with PDM3, because properties are shared among objects, there can be a problem. Consider for example the property name that has the same meaning and function whether it is applied to a person, a dog, or a company. If we want to store a default name at the property level then there may be a conflict since we are in essence specializing the shared property to one of the objects or concepts.

To cope with this problem, PDM4 uses the concept of **ideal**[1]. Every concept has an associated virtual individual instance, called its ideal, which bears all the defaults associated with the class. Thus, the ideal can be viewed as a typical individual instance of the concept. Currently, the defaults are limited to attributes. Whenever an object is built using the defindividual macro, then an ideal is created. It bears the sequence number 0. I.e., the ideal for a person will be $E-PERSON.0 if the radix for person is $E-PERSON. Actual instances will have internal ids starting at 1, i.e., $E-PERSON.1, $E-PERSON.2, etc. The ideal will behave as any individual with respect to MOSS methods. However it is not advised to temper directly with its content.

7.16.1 Setting Defaults in Relation to Concepts

Setting the ideal through defconcept

Whenever a `:default` option is used for an attribute while defining a class, then the default value associated with the option is recorded in the corresponding ideal. E.g.

```
(defconcept person
  (:att AGE (:unique)(:default 30))
  (:att NAME (:min 1)(:max 3)(:default "Dupond"))
  (:rel BROTHER PERSON))
```

[1]The name refers to the notion of idea used by Plato.

The subsequent ideal will have ”Dupond” as name and 30 as its age.

Setting the ideal through defattribute

The same result is achieved when defining the attribute separately with the `:class` option.

```
(defattribute address
  (:class person)
  (:default "Unknown"))
```

In this case however, the `:class` option must precede the default option in the list of options.

7.16.2 Setting Defaults Independently from Classes

It is possible to associate a default value directly to an attribute when defining it. Then the default will be associated directly with the attribute, independently of any class. It will stand for a typical value of this attribute. Assuming that we live in a *dark world* the default value for the concept of color could be defined as black, and stated as follows

```
(defattribute color
  (:default "Black"))
```

7.16.3 Use of Defaults

When the user ask for the value of an attribute of a given object, then the =get or =get-id method uses the following algorithm:

1. if the attribute has a local value, then return it;
2. otherwise, climb up the chain of prototypes (objects linked by an is-a property). If some object in the chain has a local value, then return this value;
3. otherwise, if the original object is an individual instance of a concept (or of several concepts), then look at the corresponding ideals for a default value. If one is found, then return it;
4. otherwise, look at the object defining the attribute. If a general default is present then return it;
5. otherwise, return nil.

7.16.4 Printing the Ideal (Typical Instance)

It is possible to print the ideal of a concept to see al the default values by using the =print-typical-instance method, e.g.,

```
(send _person '=print-typical-instance)
```

7.17 The *any* and *none* Pseudo Classes

A `*none*` marker in the $TYPE field of an object indicates an orphan?classless object. On the contrary `*any*` indicates that the object may belong to any class defined so far, thus standing for a virtual root of all IS-A lattices.

any and *none* have been declared as pseudo-classes and implemented as such. They are both of type ENTITY.

Having explicit classes could bring some simplifications in the global functioning. Indeed, orphans can be declared to be instances of the *none* class and attached to it. They would not need a special mechanism for their creation. They could also be printed easily as instances of that class. On the other hand universal methods could be attached to the *any* class as instance methods since it represents any object in the system.

Currently, what can be gained and what problems will arise from doing that is not clear. Some experience must be gained before deciding whether it is a good idea to do so.

7.18 Adding Extra Properties

A problem arises when we want to add a property to an object that is an instance of a class or several classes and the property does not belong to such classes, i.e. the object is an exception with regard to its classes. The philosophy of PDM/MOSS is to allow adding such properties.

When adding an extra property to an object, the only possibility short of adding the property at the class level is to use the generic property, since a local version of the property does not exist for the instance. When doing so, MOSS was previously issuing a warning:

```
? (send _e1 '=add-attribute-values "name" '("message 1"))
; Warning: Attribute HAS-NAME cannot be found when processing object $E-EMAIL-TEST.1
; While executing: MOSS::*0=ADD-ATTRIBUTE-VALUES
NIL
```

The new version adds the property and issuing the following warning:

```
? (send _e1 '=add-attribute-values-1 "name" '("message 1"))
; Warning: Attribute HAS-NAME does not belong to the object ($E-EMAIL-TEST.1) class(es). We ad
; While executing: MOSS::*0=ADD-ATTRIBUTE-VALUES
:done
```

7.19 Individuals Changing Concept

From version 6.1 a new method how been added to let an individual instance change concept. However this raises some issues.

The case of individuals changing concept is encountered when new information has been found about an object represented by an individual, and one wants to refine its position. E.g., we just learnt that a person is following courses, and we want to associate the individual to the concept of STUDENT rather than PERSON. When the new target concept is a sub-concept of the original concept, there is no difficulty in doing it. However, if the concept is completely different, then some of the properties of the object may not appear as properties of the new concept. But since they are properties of the object, they should be kept. The question is: should we keep such additional properties as original local properties, or should we move the property to its generic value?

Assume that we want to move a student individual to become a teacher individual. What happens to the property HAS-COURSES that was declared locally to student class? The student individual will become a teacher individual. Should we keep the local property (internally $T-STUDENT-COURSES) or use the generic property ($T-COURSES)?

When adding an extra property we use the generic property. Doing so, we could kill all the local features that were attached to the local property.

Warning: if an object has extra properties and one adds the property as a local properties of one of the classes of the object, the values that were associated with the extra property will not be reachable any longer.

Warning: the current mechanism is not compatible with multiple concept belonging.

7.20 More on Multilingual Names

Multilingual names are tricky and the corresponding operational semantics must be defined carefully.

7.20.1 Purpose

The purpose of multilingual names is to be able to designate an object by several names in different languages, and to be able to select the corresponding name or documentation according to the active language (specified by the `*language*` global variable).

7.20.2 Possible Implementations

Several ways are possible for implementing multilingual approaches, among which: (i) using versions; or (ii) using a new datatype.

Version Approach

In the version approach we associate a language with a particular version. Thus, different languages will be associated with different parallel versions. We assume that the root version uses a default language, say English.

Versions are ruled by a configuration tree, in which objects *inherit* values from above. Thus, values can be shadowed by simply specifying a set of values for a given property. For example if we specify in version 1 that the property name for the object $PNAM is "Nom de propriÈtÈ", then the version 0 value, say "Property name" is shadowed. However, the corresponding entry point PROPERTY-NAME is still reachable and points towards the right object.

Internal representation of the corresponding object:

```
(?($PNAM (0 ?Property name?)(1 ?Nom de propriÈtÈ?))?)
```

Getting the property name when in version 1 will simply return the French text.

The main advantage of this approach is that nothing needs to be changed, using a particular language amounts to select the corresponding version, namely use an in-version wrapping macro.

The main drawback of this approach is that it generates a serious conflict with the other potential uses of the versioning mechanism (temporal changes, different hypothesis). Indeed, the language issue is orthogonal to the use of versions.

Datatype Approach

The datatype approach consists in creating a new datatype whenever a multilingual text or name is needed. The value is then a vector in which entries correspond to different languages, e.g.:

```
(?Property name? ?Nom de propriÈtÈ?)
```

A better way consists in using language tags:

```
(:en ?Property name? :fr ?Nom de propriÈtÈ)
```

Internally this can be represented in many different ways including strings like:

```
?Property name@en; Nom de propriÈtÈ@fr?
```

The exact form of the internal representation does not matter as long as it allows to express text freely and functions are available to manipulate the different linguistic components.

The main advantage of this approach is that it is independent from the version mechanism.

The main drawback of the approach is that it is necessary to introduce functions and predicates associated with the new datatype.

The approach offers another interesting possibility, namely that of having all the languages available in the same environment, allowing to use any entry point to designate an object. This would not be possible in the version approach where one is restricted to the use of languages contained in the same branch of the configuration tree.

MOSS Approach

Within MOSS, I chose to extend the datatype approach with the possibility of including synonyms in the same language. The internal format would be:

```
({:name} :en ?person; individual? :fr ?personne ; individu?)
```

7.20.3 MOSS Operational Semantics

Selecting the datatype approach has several consequences. In particular, we must consider two cases: (i) creation time; and (ii) run time.

Creation Time

Creation time or loading time is the time when the internal object structures are created including, objects, entry-points, methods, functions, etc. The structures will then be used later at runtime.

At creation time one must be careful to build the right structures. Two situatios might occur: (i) we select a specific language; or (ii) we take all languages.

In the first case we can strip all multilingual information and drop the language tags. This is particularly interesting in the input files where the syntax may be simplified, the input language being specified by the value of the `*language*` global variable (e.g. `:en` or `:fr` or `:it`). Furthermore, there is a global variable named `*language-tags*` containing the set of legal languages, which allows us to introduce a specific short hand notation at creation time as follows:

```
({:name} :fr ?Londres? :* ?London?)
```

meaning that the name in French is Londres, and for all other legal languages is London. MOSS is then allowed to produce entry points corresponding to the different languages, using the necessary =make-entry methods.

If we select a specific language when loading an ontology, then the rationale is to keep only concepts and properties that have a name in the specified language. This can be done by running a trimming process that eliminates all concepts or properties that are not defined in the specified language and reducing the multilingual names to the synonyms in the specified language [2]. Globally, we obtain a projection of the ontology in the world corresponding to the specified language. This approach is not currently implemented in the MOSS/OMAS environment.

If we specify that we want all languages (`*language*` is `:all`), then we must produce all the concepts, properties, individuals, entry points in all the languages specified locally. In that case, the property names linking the objects in the input data set can be in any language.

[2] The corresponding process can be quite complex.

Run Time

We can encounter two different situations at runtime: (i) we are in a monolingual environment; (ii) we use a specific language, but the objects were created in a multilingual environment.

In the first case (monolingual environment), we have no more multilingual data. Thus, the `*language*` global variable is useless.

In the second case, we must extract for each piece of information the right label or text in the specified language. A number of multilingual primitives allow doing that.

7.20.4 Internal Coding of Names

Multilingual names pose coding problems that can only be resolved by using UNICODE (in practice UTF-8). This leads to some problems in the sense that UTF-8 is not supported by all Lisp environments (specifically MCL [3]).

[3] MCL 5.2 supports UTF-8, however one must be careful when rewriting files.

Chapter 8

Low Level Functions

Contents

Warning: MOSS 7 is an object language but implemented for efficiency reasons in terms of functional programming[1]. Thus, a number of service functions were written to deal with the object structure, names, versions, printing, etc. Such functions are documented in this document. At the beginning of the document a section deals with the macros used by the system.

The content of the document was produced automatically from the function inline documentation using the User Manual code developed by Mark Kantrowitz at CMU (1991).

[1]Not entirely true since Lisp is mostly written using CLOS (Common Lisp Obbject System).

8.1 Convention

The user should be aware of a certain number of conventions used in the system. Some are related to names, others are attached to internal obj-ids.

Elementary function names

All primitive functions start with a % sign, thus following the LISP machine LISP convention.

Some elementary functions are very primitive and somehow dangerous to use, they start with a double %% sign.

Some functions contain the string tp or sp; meaning they apply to attributes or to relations. This comes from a time when attributes were called terminal-properties and relations structural properties. Similarly, class or entity is used to refer to concepts and instances to individuals.

Predicates

Each function in the following set can apply to any object in the system. Predicates end with a question mark following the Scheme convention.

Example: %terminal-property?

Context

Most functions must execute within a *context* (or *version*), i.e. one of the states of the knowledge base corresponding to a node of the version graph. When the context is not passed as an argument to the function, and if it is needed, then the default context `*context*` is used. A version graph recorded in the global variable `*version-graph*` is used to impose the adequate structure onto the data.

Method names

By convention method names start with the = sign. Although this is in no way compulsory it is a good practice to avoid confusion with other kind of variables.

property names

External property names acquire the HAS- prefix, so that they can be distinguished from the class names that don't.

examples: HAS-NAME, HAS-DOCUMENTATION

Inverse property names acquire the IS- prefix and -OF suffix unless a special name is specified for them.

Example: IS-ENTITY-NAME-OF

Object ids

Object-ids are assigned by the system and are intended to be used only by the system functions. As a general rule one should not rely on such names, since the system may reassign them to different entities if the system is rebooted. They are only used to distinguish among various objects.

In the MOSS kernel they have names like $ENT, $EPT, $SUC, $CTR, etc. and are interned into the :moss package.

Some instances like methods have more complicated names like: $FN.34

User internal object-ids have systematic naming: (i) classes: $E-XXX, e.g. $E-PERSON, from the canonical external name; (ii) attributes: $T-XXX; (iii) relations: $S-XXX; (iv) individuals: <class-id>.nn, e.g. $E-PERSON.32. The corresponding symbols are interned in the user?s application package (most often :common-lisp-user).

Property ids

A special mention for property ids.

Properties are defined not as a single object but as an object tree. Namely, when a property is defined, a generic property is created, e.g.

```
(defattribute AGE)
```

creates a generic property named $T-NAME

If the property is defined in the context of a class or later added to a class, then a local version is created in addition to the generic version, e.g.

```
(defconcept PERSON (:att age))
```

creates (if needed) a generic property named $T-NAME and a local property $T-PERSON-NAME as a sub-property of the generic property. Both properties share the same HAS-NAME external name.

The mechanism applies also to relations.

8.2 Macros and Global Variables

The macros documented here can be grouped into several sets. Some macros are related to the object implemented format, some are predicates, some are offered to simplify the declaration of kernel classes or properties or methods, others are quite obscure service functions. In this section they are grouped according to their use. Later on, they are presented in alphabetical order

8.2.1 Defining Macros

They are intended to facilitate the definition of MOSS objects.

```
;;; DEFATTRIBUTE (name &rest option-list)                               [MACRO]
;;;
;;; %DEFCLASS (multilingual-name &rest option-list)                     [MACRO]
;;;
;;; DEFCHAPTER (&rest option-list)                                      [MACRO]
;;;
;;; DEFCONCEPT (multilingual-name &rest option-list)                    [MACRO]
;;;
;;; %DEFCONCEPT (multilingual-name &rest option-list)                   [MACRO]
;;;
;;; %DEFGENERICTP (name &rest option-list)                              [MACRO]
;;;    see %make-generic-tp doc
;;;
;;; DEFINDIVIDUAL (name &rest option-list)                              [MACRO]
;;;
;;; DEFINSTANCE (name &rest option-list)                                [MACRO]
;;;
;;; %DEFINSTANCE (name &rest option-list)                               [MACRO]
```

```
;;;
;;; DEFINSTMETHOD (name selector arg-list &rest body)                  [MACRO]
;;;
;;; DEFMOSSINSTMETHOD (name selector arg-list &rest body)              [MACRO]
;;;
;;; DEFMOSSOWNMETHOD (name selector arg-list &rest body)               [MACRO]
;;;
;;; DEFMOSSUNIVERSALMETHOD (name arg-list &rest body)                  [MACRO]
;;;
;;; DEFOBJECT (&rest option-list)                                      [MACRO]
;;;
;;; %DEFOBJECT (&rest option-list)                                     [MACRO]
;;;
;;; DEFONTOLOGY (&rest option-list)                                    [MACRO]
;;;
;;; DEFOWNMETHOD (name selector arg-list &rest body)                   [MACRO]
;;;
;;; %DEFOWNMETHOD (name selector arg-list &rest body)                  [MACRO]
;;;
;;; %DEFMETHOD (name selector arg-list &rest body)                     [MACRO]
;;;
;;; DEFRELATION (name class suc &rest option-list)                     [MACRO]
;;;
;;; DEFSECTION (&rest option-list)                                     [MACRO]
;;;
;;; %DEFSP (name class suc &rest option-list)                          [MACRO]
;;;
;;; %DEFSYSVAR (name doc value)                                        [MACRO]
;;;
;;; %DEFSYSVARNAME (sysvar-name)                                       [MACRO]
;;;
;;; %DEFTP (name &rest option-list)                                    [MACRO]
;;;
;;; DEFVIRTUALCONCEPT (multilingual-name &rest option-list)            [MACRO]
;;;
;;; %DEFVIRTUALCONCEPT (multilingual-name &rest option-list)           [MACRO]
;;;
;;; %DEFUNIVERSAL (name arg-list &rest body)                           [MACRO]
;;;
;;; DEFUNIVERSALMETHOD (name arg-list &rest body)                      [MACRO]
;;;
```

8.2.2 Printing Debugging and Tracing Macros

```
;;; DFORMAT (cstring &rest args)                                       [MACRO]
;;;
;;; VFORMAT (cstring &rest args)                                       [MACRO]
;;;
;;; TRFORMAT (fstring &rest args)                                      [MACRO]
;;;    debugging macro used locally explicitely temporarily
```

```
;;;
;;; RFORMAT (limit &rest ll)                                          [MACRO]
;;;    takes a text and limits its length. Because the silly Pascal strings
;;;    must be ~ less than some length.
;;;    Arguments:
;;;     limit: max length
;;;     ll: regular arguments to a text.
;;;
;;; TFORMAT (format-string &rest arg-list)                            [MACRO]
;;;    used to simplify global trace
;;;
```

8.2.3 Macros Dealing with Names

```
;;; %MAKE-NAME-FOR-CLASS (name &key package)                          [MACRO]
;;;    build the class name, e.g. SECTION-TITLE.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-CLASS-PROPERTY (prop-ref class-ref &key package)    [MACRO]
;;;    build the property name symbol, e.g. HAS-PERSON-FIRST-NAME.
;;;    Argument:
;;;    prop-ref: symbol or (multilingual) string
;;;    class-ref: specifies the class
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-CONCEPT (name &key package)                        [MACRO]
;;;    build the concept name, e.g. SECTION-TITLE.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-ENTRY (name &key package)                          [MACRO]
;;;    build an entry name, e.g. SECTION-TITLE.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-INVERSE-PROPERTY (name &key package)               [MACRO]
;;;    build the inverse property name, e.g. IS-FIRST-NAME-OF.
;;;    Argument:
```

```
;;;    name: symbol or (multilingual) string
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-INSTANCE-METHOD-FUNCTION (name class-id              [MACRO]
;;;                                          &key (context *context*)
;;;                                          package)
;;;    build the instance method function name, e.g. ~
;;;         $E-PERSON=I=0=PRINT-SELF.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    class-id: class id that owns the method
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-OWN-METHOD-FUNCTION (name obj-id                     [MACRO]
;;;                                     &key (context *context*)
;;;                                     package)
;;;    build the own method function name, e.g. $E-PERSON=S=0=PRINT-SELF.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    class-id: id of the object that owns the method
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-PROPERTY (name &key package)                         [MACRO]
;;;    build the property name, e.g. HAS-FIRST-NAME.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    package (opt): package if different from current
;;;    Return:
;;;    interned symbol in the current package.
;;;
;;; %MAKE-NAME-FOR-UNIVERSAL-METHOD-FUNCTION (name                      [MACRO]
;;;                                           &key context package)
;;;    build the universal method function name, e.g. *0=PRINT-UNIVERSAL.
;;;    Argument:
;;;    name: symbol or (multilingual) string
;;;    context (key): default current
;;;    package (key): package if different from current
;;;    Return:
;;;    interned symbol in the specified package.
;;;
;;; %MAKE-NAME-FOR-VARIABLE (name &key package)                         [MACRO]
;;;    Builds an external variable name starting with underscore.
;;;    Argument:
;;;    name: variable  name
```

```
;;;    Returns:
;;;    the name starting with an underscore
;;;
;;; %MAKE-NAME-STRING-FOR-CONCEPT (name)                                [MACRO]
;;;    Cooks up a name string from the name of a concept.
;;;    Argument:
;;;    name: a symbol or (multilingual) string naming the concept
;;;    Return:
;;;    a string naming the concpet
;;;
;;; %MAKE-NAME-STRING-FOR-CONCEPT-PROPERTY (name-string class-ref)      [MACRO]
;;;    Builds an external property name string, starting with HAS-
;;;    Arguments:
;;;    name: a multilingual name naming the property given by the user as
;;;    an ~ argument to m-deftp for example (m-deftp (:fr "titre de
;;;    l'ouvrage" :en "book title") ...) Return:
;;;    a string naming the property, e.g. "HAS-TITLE"
;;;
;;; %MAKE-NAME-STRING-FOR-INVERSE-PROPERTY (name)                       [MACRO]
;;;    Cooks up an inverse property string from the name of the property by
;;;    sticking ~ IS- and -OF an each side. Remember the inverse property
;;;    has no semantic ~ content. It simply provides the possibility of
;;;    following the arcs in the ~ reverse direction.
;;;    Argument:
;;;    name: a symbol or string naming the direct property
;;;    Return:
;;;    a string naming the inverse property
;;;
;;; %MAKE-NAME-STRING-FOR-METHOD-FUNCTION (class-id method-type         [MACRO]
;;;                                        context method-name)
;;;    Builds an internal name to store compiled methods, e.g.
;;;    $CTR=I=2=summary ~ or $E-PERSON=S=1=print-self where 1 is the context
;;;    value. Arguments:
;;;    class-id: identifier of the class
;;;    method-type: :instance (I) or own (S)
;;;    context: context
;;;    method-name: name of the method
;;;    Returns:
;;;    a symbol with package that of the method name.
;;;
;;; %MAKE-NAME-STRING-FOR-PROPERTY (name-string)                        [MACRO]
;;;    Builds an external property name string, starting with HAS-
;;;    Arguments:
;;;    name: a multilingual name naming the property given by the user as
;;;    an ~ argument to m-deftp for example (m-deftp (:fr "titre de
;;;    l'ouvrage" :en "book title") ...) Return:
;;;    a string naming the property, e.g. "HAS-TITLE"
;;;
;;; %MAKE-NAME-STRING-FOR-RADIX (name &optional (type ""))              [MACRO]
;;;    build a radix name string for a property or for a class. Tries to
```

```
;;;    build a unique name ~ using the first 3 letters of the name. Error if
;;;    the symbol already exists. Arguments:
;;;    name: e.g. "SIGLE"
;;;    type (opt): e.g. "T " (for terminal property)
;;;    Return:
;;;    e.g. $T-SIG or error if already exists
;;;
;;; %MAKE-NAME-STRING-FOR-UNIVERSAL-METHOD-FUNCTION (context name)      [MACRO]
;;;    Builds an internal name to store compiled methods.
;;;    Arguments:
;;;    context: context
;;;    name: name of the universal method
;;;    Returns:
;;;    a symbol with package that of the method name.
;;;
;;; %MAKE-NAME-STRING-FOR-VARIABLE (name)                               [MACRO]
;;;    Builds an internal name string for a variable.
;;;    Arguments:
;;;    name: string, symbol or multilingual name
;;;    Returns:
;;;    a string.
;;;
;;; %MKNAME (ref type &key package)                                     [MACRO]
;;;    if ref is a symbol return symbol, otherwise call~
;;;         %make-name-for-<type>.
;;;    Arguments:
;;;    ref: a symbol or (multilingual) string
;;;    type: type of name to recover (keyword)
;;;    package (key): default current
;;;    Expansion:
;;;    (%make-name-for-<type> ref . more-args)
;;;
```

8.2.4 Macros Dealing with Ids

```
;;; %MAKE-ID-FOR-CLASS (name &key package)                              [MACRO]
;;;    function to cook up default entity ids: $E-<name>
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;         if not there into the current package.
;;;    Argument:
;;;    name: symbol or (multilingual) to be used for radix
;;;    package (key): default current
;;;    Return:
;;;    a symbol specifying the class, e.g. $E-PERSON
;;;
;;; %MAKE-ID-FOR-VIRTUAL-CLASS (name &key package)                      [MACRO]
;;;    function to cook up default entity ids: $E-<name>
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
```

```
;;;         if not there into the current package.
;;;    Argument:
;;;    name: symbol or (multilingual) to be used for radix
;;;    package (key): default current
;;;    Return:
;;;    a symbol specifying the class, e.g. $E-PERSON
;;;
;;; %MAKE-ID-FOR-CLASS-SP (name class-name &key package)                [MACRO]
;;;    function to cook up a default relation id.
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;         if not there into the current package.
;;;    Argument:
;;;    name: symbol or (multilingual) to be used for radix
;;;    package (key): default current
;;;    Return:
;;;    a symbol specifying the class, e.g. $S-<class-name>-<name>
;;;
;;; %MAKE-ID-FOR-CLASS-TP (name class-name &key package)                [MACRO]
;;;    function to cook up default property ids.
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;         if not there into the current package.
;;;    Argument:
;;;    name: symbol or (multilingual) to be used for radix
;;;    package (key): default current
;;;    Return:
;;;    a symbol specifying the class, e.g. $T-<class-name>-<name>
;;;
;;; %MAKE-ID-FOR-CONCEPT (name &key package)                            [MACRO]
;;;    see %make-id-for-class
;;;
;;; %MAKE-ID-FOR-VIRTUAL-CONCEPT (name &key package)                    [MACRO]
;;;    see %make-id-for-virtual-class
;;;
;;; %MAKE-ID-FOR-COUNTER (class-id)                                     [MACRO]
;;;    makes an ID for a counter, e.g. $E-PERSON.CTR in class-id package.
;;;    Arguments:
;;;    class-id: id of the class to which the counter belongs
;;;    Returns:
;;;    the counter id.
;;;
;;; %MAKE-ID-FOR-IDEAL (class-id)                                       [MACRO]
;;;    makes an ID for the class ideal, e.g. $E-PERSON.0 in class-id
;;;    package. Arguments:
;;;    class-id: class ID
;;;    Return:
;;;    a symbol that is the ID of the ideal.
;;;
;;; %MAKE-ID-FOR-INSTANCE (model-id value &key context package)         [MACRO]
```

```
;;;    Builds internal keys for instances using the radix of a class; e.g. ~
;;;         $E-PERSON.34 (34 is the value obtained from the class counter).
;;;    The package of the new symbol is that of the class.
;;;    Arguments:
;;;    model-id: identifier of the class
;;;    value: value of the counter
;;;    context (key): default current
;;;    package (key): ignored
;;;    Returns:
;;;    an identifier obtained by applying make-name.
;;;
;;; %MAKE-ID-FOR-INSTANCE-METHOD (context value &key package)           [MACRO]
;;;    Builds an internal id for an own-method: e.g. $FN.34
;;;    Arguments:
;;;    context: context
;;;    value: counter for methods
;;;    package (key): default current
;;;    Return:
;;;    method id.
;;;
;;; %MAKE-ID-FOR-INVERSE-PROPERTY (id)                                  [MACRO]
;;;    Builds an inverse property id from a direct one. The function is
;;;    useful  ~ only for MOSS kernel. Indeed, for applications, inverse
;;;    properties are ~ instances of the $EIL class, and have $EIL.nn as
;;;    internal name. Arguments:
;;;    id: identifier of the direct property
;;;    Return:
;;;    symbol for the inverse property id.
;;;
;;; %MAKE-ID-FOR-ORPHAN (context value &key package)                    [MACRO]
;;;    Builds an internal id for an orphan object: e.g. $3-5 where 3 is ~
;;;         the context value, 5 the system's counter value.
;;;    Arguments:
;;;    context: context
;;;    value: counter for orphans
;;;    Return:
;;;    orphan id.
;;;
;;; %MAKE-ID-FOR-OWN-METHOD (context value &key package)                [MACRO]
;;;    Builds an internal id for an own-method: e.g. $FN.34
;;;    Arguments:
;;;    context: context
;;;    value: counter for methods
;;;    package (key): default current
;;;    Return:
;;;    method id.
;;;
;;; %MAKE-ID-FOR-SP (name &key package)                                 [MACRO]
;;;    function to cook up default property ids:
;;;    If name is a symbol then the symbol is interned into the symbol's
```

```
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;        if not there into the current package.
;;;    Arguments:
;;;    name: symbol, string or mln
;;;    package (key): default current
;;;    Return:
;;;    a symbol anem $S-<name>
;;;
;;; %MAKE-ID-FOR-TP (name &key package)                               [MACRO]
;;;    function to cook up default property ids: $T-<name>
;;;    If name is a symbol then the symbol is interned into the symbol's
;;;    ~ package, otherwise it is interned into the specified package, or ~
;;;        if not there into the current package.
;;;    Arguments:
;;;    name: symbol or (multilingual) string
;;;    package (key): default current
;;;    Return:
;;;    a symbol anem $T-<name>
;;;
;;; %MAKE-ID-FOR-UNIVERSAL-METHOD (context value &key package)        [MACRO]
;;;    Builds an internal id for an own-method: e.g. $UNI.34
;;;    Arguments:
;;;    context: context
;;;    value: counter for methods
;;;    package (key): default current
;;;    Return:
;;;    method id.
;;;
```

8.2.5 Program Control

```
;;; CATCH-ERROR (msg &rest body)                                      [MACRO]
;;;    intended to catch errors due to poor MOSS syntax. Prints in active
;;;    window ~ using mformat unless msg is empty
;;;    Note: body should not return a string (considered as an error
;;;    message) Arguments:
;;;    msg: something like "while executing xxx"
;;;    body (rest): exprs to execute
;;;    Return:
;;;    the value of executing body if no error, nil and prints a message
;;;    when error
;;;
;;; CATCH-SYSTEM-ERROR (msg &rest body)                               [MACRO]
;;;
;;; MTHROW (control-string &rest args)                                [MACRO]
;;;    print a message and throws the message string to :error
;;;
;;; TERROR (cstring &rest args)                                       [MACRO]
;;;    throws an error message back to an :error catch. Prints message if
;;;    *verbose*
```

```
;;;
;;; TWARN (cstring &rest args)                                    [MACRO]
;;;    inserts a message into the *error-message-list* making sure to return
;;;    nil
;;;
;;; VERBOSE-THROW (label &rest format-args)                        [MACRO]
;;;    when *verbose* is T, prints a warning before throwing the error
;;;    message ~ string to the label.
;;;    Arguments:
;;;    label: a label to which to throw (usually :error)
;;;    format-args: arguments to build a message string using format nil
;;;    Return:
;;;    throws to the label the message string.
;;;
;;; VERBOSE-WARN (&rest format-args)                               [MACRO]
;;;
;;; WHILE (condition &rest body)                                   [MACRO]
;;;
```

8.2.6 Environment

```
;;; IN-VERSION (%_context &rest ll)                                [MACRO]
;;;
;;; WITH-CONTEXT (%_context &rest ll)                              [MACRO]
;;;
;;; WITH-PACKAGE (%_package &rest ll)                              [MACRO]
;;;
```

8.2.7 Low Level or Compatibility Macros and Functions

Such macros or functions are used for facilitating programming or for upward compatibility reasons.

```
;;; ASSREMPROP (prop ll)                                           [MACRO]
;;;
;;; GETAV (key list)                                               [MACRO]
;;;
;;; GETV (prop ll)                                                 [MACRO]
;;;
;;; INTER (&rest ll)                                               [MACRO]
;;;
;;; NASSREMPROP (prop ll)                                          [MACRO]
;;;
;;; NCONS (arg)                                                    [MACRO]
;;;
;;; PUTPROP (xx val prop)                                          [MACRO]
;;;
;;; ALISTP (ll)                                                 [FUNCTION]
;;;
;;; ALL-ALIKE? (ll)                                             [FUNCTION]
;;;    T if all elements of the list are the same
;;;
```

```
;;; ALL-DIFFERENT? (ll)                                         [FUNCTION]
;;;    T if all elements of the list are different
;;;
;;; INTERSECT-SETS (list)                                       [FUNCTION]
;;;
;;; LAMBDAP (symbol)                                            [FUNCTION]
;;;    check if symbol is a lambda list
;;;
;;; LOB-GET (obj-id prop-id)                                    [FUNCTION]
;;;
;;; LOB-TYPEP (xx)                                              [FUNCTION]
;;;
;;; MEXP (form)                                                 [FUNCTION]
;;;
;;; MAKE-NAME (&rest arg-list)                                  [FUNCTION]
;;;
;;; MAKE-VALUE-STRING (value &optional (interchar '#\.))        [FUNCTION]
;;;    Takes an input string, removing all leading and trailing blanks,
;;;    replacing ~ all substrings of blanks and simple quote with a single
;;;    underscore, and ~ building a new string capitalizing all letters. The
;;;    function is used ~ to normalize values for comparing a value with an
;;;    external one while ~ querying the database.
;;;         French accentuated letter are replaced with unaccentuated
;;;    capitals.
;;;
;;; MFORMAT (control-string &rest args)                         [FUNCTION]
;;;    prints to the output specified by output and a trace into the
;;;    listener. Arguments:
;;;    output: stream (main be a window pane
;;;    control-string: format control string
;;;    args (rest): args if any
;;;    Returns:
;;;    output-string (to be used in throw eventually)
;;;
;;; MGF-CLOSE "()"                                              [FUNCTION]
;;;    Closes the currently opened database
;;;
;;; MGF-END-TRANSACTION "()"                                    [FUNCTION]
;;;
;;; MGF-ERRORP "()"                                             [FUNCTION]
;;;    Returns the error code set by MGF functions.
;;;
;;; MGF-LOAD (xx)                                               [FUNCTION]
;;;    noop as for now
;;;
;;; MGF-OPEN (filename)                                         [FUNCTION]
;;;    Opens a specific database and puts its name into the *base* global
;;;    variable
;;;
;;; MGF-START-TRANSACTION "()"                                  [FUNCTION]
```

```
;;;    In a multiuser environment prevents other users to access the MGF
;;;    file.
;;;
;;; MGF-STORE (key value)                                           [FUNCTION]
;;;    stores a given key and value into the database
;;;
;;; MGF-UPDATE (key value)                                          [FUNCTION]
;;;    stores a given key and value into the database replacing old value if
;;;    any.
;;;
;;; NO-P (word)                                                     [FUNCTION]
;;;    Returns some non nil value if arg is noor non or some substring
;;;
;;; NALIST-REPLACE-PV (obj-id prop-id value &aux obj-l)             [FUNCTION]
;;;
;;; NPUTV (prop-id value ll)                                        [FUNCTION]
;;;    nputv is defined for compatibility with UTC-Lisp. It is a destructive
;;;    function it adds a value to the list of values corresponding to a
;;;    property without duplicating it
;;;
;;; NPUTV-LAST (prop val ll)                                        [FUNCTION]
;;;    nputv-last is defined for compatibility with UTC-Lisp. It adds a
;;;    value to the list of values associated with a property in an a-list
;;;    at the last position. Destructive function.
;;;
;;; REF (symbol &optional show-file)                                [FUNCTION]
;;;    traces in what function a specific symbol appears by examining the
;;;    code of the ~ function in the corresponding file.
;;;
;;; APPEARS-IN? (symbol expr)                                       [FUNCTION]
;;;
;;; YES-P (word)                                                    [FUNCTION]
;;;    Returns some non nil value if arg is yes or oui or some substring
```

8.2.8 Object Structure

```
;;; MOSS-SYMBOL? (obj-id)                                           [FUNCTION]
;;;    check whether symbol is defined in the moss package
;;;
;;; %%RESOLVE (id context)                                             [MACRO]
;;;    resolves references, pseudo objects with a $REF property, replacing
;;;    the ~ specified id with the id pointed to in the specified context.
;;;    If unbound or actual object, then returns the input id.
;;;    Arguments:
;;;    id: symbol containing the object id
;;;    context (opt): context (default current)
;;;    Return:
;;;    id or id pointed to
;;;
```

8.2.9 Global Variables

```
;;; *APPLICATION-NAME* ("moss-test")                                [PARAMETER]
;;;    default disk file
;;;
;;; *CONTEXT* (0)                                                   [PARAMETER]
;;;    number of the current ontology version
;;;
;;; *LANGUAGE* (:en)                                                [PARAMETER]
;;;    default language is English
;;;
;;; *VERSION-GRAPH* ('((0)))                                        [PARAMETER]
;;;    configuration lattice
;;;
;;; *SENDER* (nil)                                                  [PARAMETER]
;;;    sender of MOSS message
;;;
;;; *ANSWER* ("answer returned by a moss method")                    [VARIABLE]
;;;
;;; *SELF* (nil)                                                    [PARAMETER]
;;;    id of object that received the MOSS message
;;;
;;; *METHOD-NAME* ("name of the method being executed")             [PARAMETER]
;;;
;;; *EXPORT-ENTRY-SYMBOLS* (t)                                      [PARAMETER]
;;;    MOSS system exports entry symbols
;;;
;;; *DEBUG* (nil)                                                   [PARAMETER]
;;;    debug flag triggering the dformat macro
;;;
;;; *DELIMITERS* ('(#\space #\, #\' #\.))                           [PARAMETER]
;;;    characters delimiting a word
;;;
;;; *TRACED-AGENT* (nil)                                            [PARAMETER]
;;;    omas agent being traced
;;;
;;; *SYSTEM-ENTITIES* (nil)                                          [VARIABLE]
;;;    record the list of MOSS entities (classes)
;;;
;;; *WARNING-FOR-OWN-METHODS* (nil)                                 [PARAMETER]
;;;    flag for warning messages
;;;
;;; *VERBOSE* (nil)                                                  [VARIABLE]
;;;    trace error messages in warnings
;;;
;;; *ALLOW-FORWARD-REFERENCES* (nil)                                [PARAMETER]
;;;    for relatiosns to an undefined object
;;;
;;; *ALLOW-FORWARD-INSTANCE-REFERENCES* (nil)                       [PARAMETER]
;;;    for relatiosns to an undefined instance
```

```
;;;
;;; *DEFERRED-ACTIONS* (nil)                                        [PARAMETER]
;;;    list of actions deferred until the end of the file
;;;
;;; *DEFERRED-INSTANCE-CREATIONS* (nil)                             [PARAMETER]
;;;    list of instance creation actions
;;;
;;; *DEFERRED-DEFAULTS* (nil)                                       [PARAMETER]
;;;    list of default creation actions
;;;
;;; *BASE* (nil)                                                     [VARIABLE]
;;;    Name of the currently opened base
;;;
;;; *DISK* (nil)                                                     [VARIABLE]
;;;    a-list simulating a disk file
;;;
;;; *LEFT-MARGIN* (0)                                                [VARIABLE]
;;;
;;; *LOB* (nil)                                                      [VARIABLE]
;;;    LOB object for handling update programs. Initialized in the LOB
;;;    module.
;;;
;;; *MGF-ERROR* (nil)                                                [VARIABLE]
;;;
;;; *MOSS-STAND-ALONE* (t)                                           [VARIABLE]
;;;    indicates whether MOSS is standalone or not
;;;
;;; *MOSS-TRACE-LEVEL* (0)                                           [VARIABLE]
;;;    Column number for starting the trace print out.
;;;
;;; *PRETTY-TRACE* (nil)                                             [VARIABLE]
;;;
;;; *STRING-HYPHEN* ("-")                                            [VARIABLE]
;;;
;;; *MOSS-SYSTEM* (nil)                                              [VARIABLE]
;;;
;;; *TRANSACTION-NUMBER* (nil)                                       [VARIABLE]
;;;    Used by LOB
;;;
;;; *CONVERSATION* (nil)                                            [PARAMETER]
;;;    current-conversation; not used for OMAS
;;;
;;; *MOSS-CONVERSATION* (nil)                                       [PARAMETER]
;;;    current-conversation; not used for OMAS
;;;
;;; *DIALOG-VERBOSE* (nil)                                          [PARAMETER]
;;;    trace of dialog flag
;;;
;;; *TRANSITION-VERBOSE* (nil)                                      [PARAMETER]
;;;    trace dialog transitions flag
```

```
;;;
;;; *MOSS-ENGINE-LOADED* (nil)                                [VARIABLE]
;;;    indicates that the send mechanism is loaded
;;;
;;; *CACHE-METHODS* (nil)                                     [VARIABLE]
;;;    Flag that allows to cache methods onto p-list of ~
;;;                                                      classes and
;;;    objects. Turned off for debugging purposes
;;;
;;; *LEXICOGRAPHIC-INHERITANCE* (t)                           [VARIABLE]
;;;    Disable fancy inheritance scheme in favor ~
;;;
;;;    of a standard lexicographic scheme
;;;
;;; *USER* ('anonymous)                                       [VARIABLE]
;;;    The identity of the user of the system.
;;;
;;; *TRACE-FLAG* (nil)                                        [VARIABLE]
;;;    Flag for toggling trace of messages on and off.
;;;
;;; *TRACE-MESSAGE* (nil)                                     [VARIABLE]
;;;    Flag for tracing all messages to all objects.
;;;
;;; *MOSS-WINDOW* (nil)                                      [PARAMETER]
;;;    MOSS interface
;;;
;;; *MOSS-OUTPUT* (t)                                        [PARAMETER]
;;;    output channel: t or MOSS window
;;;
;;; *MOSS-INPUT* (t)                                         [PARAMETER]
;;;    input channel: t or MOSS window
;;;
;;; *ALLOWED-RESTRICTION-OPERATORS* ('(:type :not-type :unique      [VARIABLE]
;;;                                 :one-of :exists :forall :not
;;;                                 :value :between :outside
;;;                                 :filter :same :different :min
;;;                                 :max :unique))
;;;    allowed operators used to restrict the value of a property
;;;
;;; *LANGUAGE-TAGS* ('(:en :fr :it :pl))                     [PARAMETER]
;;;    allowed languages
;;;
```

8.3 Functions

In this section, functions have been grouped by features.

8.3.1 Resetting the environment

```
;;; %RESET (&key (context *context* there?))                 [FUNCTION]
```

```
;;;    Removes all classes and instances of an application, making the
;;;    corresponding ~ symbols unbound. Protects all data defined in the
;;;    kernel (moss  package). Arguments:
;;;    none
;;;    Return:
;;;    t
;;;
```

8.3.2 Predicates

```
;;; %ANCESTOR? (obj1 obj2 &key (prop-id '$is-a)                    [FUNCTION]
;;;             (context *context*))
;;;    Checks if obj2 is equal to obj1, or belongs to the transitive closure
;;;    of obj1 ~ using property prop-id.
;;;    Arguments:
;;;    obj1: first object
;;;    obj2: second object
;;;    prop-id: property to follow (default is $IS-A)
;;;    context (opt): context (default current)
;;;
;;; %CLASSLESS-OBJECT? (object-id &key (context *context*))         [FUNCTION]
;;;    Checks if the entity is a classless object, i.e. its type is *none*
;;;    Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    Return:
;;;    T if OK, NIL otherwise.
;;;
;;; %ENTRY? (object-id &key (context *context*)                     [FUNCTION]
;;;          (package *package* package-here?))
;;;    Checks if the id is that of an entry point.
;;;    Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    package (opt): if obj-id is a symbol, then that of obj-id default
;;;    current Returns:
;;;    nil or something not meaningful
;;;
;;; %IS-A? (obj1 obj2 &key (context *context*))                     [FUNCTION]
;;;    Checks if obj1 is equal to obj2, or obj-2 belongs to the transitive
;;;    closure of obj1~ using property $IS-A.
;;;    Arguments:
;;;    obj1: first object
;;;    obj2: second object
;;;    context (opt): context (default current)
;;;
;;; %IS-ATTRIBUTE? (prop-id &key (context *context*))               [FUNCTION]
;;;    Checks if the identifier points towards an attribute
;;;    Arguments:
;;;    prop-id: object identifier
```

```
;;;    context (opt): context (default current)
;;;    Return:
;;;    T or nil
;;;
;;; %IS-ATTRIBUTE-MODEL? (object-id &key (context *context*))        [FUNCTION]
;;;    Checks if the object is $EPT or some sub-type.
;;;    Arguments:
;;;    object-id: object identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    T or nil
;;;
;;; %IS-CLASS? (object-id &key (context *context*))                  [FUNCTION]
;;;    Checks if the object is a class, i.e. if its type is $ENT or a
;;;    subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-CONCEPT? (object-id &key (context *context*))                [FUNCTION]
;;;    Checks if the object is a concept (same as class), i.e. if its type
;;;    is $ENT or a subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-COUNTER-MODEL? (object-id &key (context *context*))          [FUNCTION]
;;;    Checks if the object is $CTR or some sub-type.
;;;    Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-ENTITY-MODEL? (object-id &key (context *context*))           [FUNCTION]
;;;    Checks if the object is $ENT or some sub-type.
;;;
;;; %IS-ENTRY-POINT-INSTANCE? (object-id &key (context *context*))   [FUNCTION]
;;;    Checks if the object is an instance of entry-point.
;;;
;;; %IS-GENERIC-PROPERTY-ID? (object-id &key (context *context*))    [FUNCTION]
;;;    Checks if the object is $EPR and is not a subproperty of some other
;;;    property.
;;;
;;; %IS-INSTANCE-OF? (object-id class-id)                            [FUNCTION]
;;;    Checks if the object is an instance of class id or of one of its
;;;    children. Arguments:
;;;    object-id: id of the object too test
```

```
;;;    class-id: id of the class
;;;    Return:
;;;    t if true nil otherwise.
;;;
;;; %IS-INVERSE-LINK-MODEL? (object-id &key (context *context*))     [FUNCTION]
;;;    Checks if the object is $EIL or some sub-type.
;;;
;;; %IS-INVERSE-PROPERTY? (xx &key (context *context*))              [FUNCTION]
;;;    Checks if the id points towards an inverse property, or to a property
;;;    containing $EIL in its transitive closure along $IS-A.
;;;
;;; %IS-INVERSE-PROPERTY-REF? (xx &key (context *context*))          [FUNCTION]
;;;    Checks if the string is the reference of an inverse property, i.e. a
;;;    string ~ starting with the > symbol whose name is that of a property.
;;;    Argument:
;;;    xx: string
;;;    Return:
;;;    the inverse property-id or nil.
;;;
;;; %IS-METHOD? (xx &key (context *context*))                        [FUNCTION]
;;;    Checks if the id points towards a method - Optional arg: context
;;;
;;; %IS-METHOD-MODEL? (object-id &key (context *context*))           [FUNCTION]
;;;    Checks if the object is $EFN or some sub-type.
;;;
;;; %IS-MODEL? (object-id &key (context *context*))                  [FUNCTION]
;;;    Checks whether an object is a model (class) or not - Models ~
;;;         are objects that have properties like $PT or $PS or $RDX or
;;;    $CTRS ~ or whose type is in the transitive closure of $ENT along the
;;;    $IS-A.OF link.
;;;
;;; %IS-ORPHAN? (object-id &key (context *context*))                 [FUNCTION]
;;;    Short synonym for %classless-object?
;;;
;;; %IS-RELATION? (prop-id &key (context *context*))                 [FUNCTION]
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-RELATIONSHIP? (prop-id &key (context *context*))             [FUNCTION]
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-RELATION-MODEL? (object-id &key (context *context*))         [FUNCTION]
;;;    checks if the object is $EPS or some sub-type
;;;
;;; %IS-STRUCTURAL-PROPERTY? (prop-id &key (context *context*))      [FUNCTION]
;;;    Checks if the identifier points towards a structural property -
;;;     Optional arg: context
;;;
;;; %IS-STRUCTURAL-PROPERTY-MODEL? (object-id                        [FUNCTION]
;;;                                 &key (context *context*))
;;;    Checks if the object is $EPS or some sub-type.
```

```
;;;
;;; %IS-SYSTEM? (object-id &key (context *context*))                [FUNCTION]
;;;    Checks if the id points towards a system - Optional arg: context
;;;
;;; %IS-SYSTEM-MODEL? (object-id &key (context *context*))          [FUNCTION]
;;;    Checks if the entity is a model, and belongs to the MOSS package.~
;;;         Arguments:
;;;    object-id: identifier of object to test
;;;    context (opt): context default current
;;;    Return:
;;;    nil or t
;;;
;;; %IS-TERMINAL-PROPERTY? (prop-id &key (context *context*))       [FUNCTION]
;;;    Checks if the identifier points towards a terminal property -
;;;     Optional arg: context
;;;
;;; %IS-TERMINAL-PROPERTY-MODEL? (object-id                         [FUNCTION]
;;;                               &key (context *context*))
;;;    Checks if the object is $EPT or some sub-type.
;;;
;;; %IS-UNIVERSAL-METHOD? (xx &key (context *context*))             [FUNCTION]
;;;    Checks if the id points towards a universal method - Optional arg:
;;;    context
;;;
;;; %IS-UNIVERSAL-METHOD-MODEL? (object-id                          [FUNCTION]
;;;                              &key (context *context*))
;;;    Checks if the object is $UNI or some sub-type.
;;;
;;; %IS-VALUE-TYPE? (value value-type)                              [FUNCTION]
;;;    Checks that a value has the right type.
;;;    Arguments:
;;;    value: value to check
;;;    value-type: one of the XSD types (birkkkhh!)
;;;    Returns:
;;;    value if the right type, nil otherwise.
;;;
;;; %IS-VARIABLE-NAME? (ref)                                        [FUNCTION]
;;;    Checks if the refernce is a variable name, i.e. a symbol starting
;;;    with _ (underscore)
;;;
;;; %IS-VIRTUAL-CONCEPT? (object-id &key (context *context*))       [FUNCTION]
;;;    Checks if the object is a concept (same as class), i.e. if its type
;;;    is $ENT or a subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %PCLASS? (input class-ref)                                      [FUNCTION]
;;;    tests if input belongs to the transitive closure of class-ref. Only
```

```
;;;    works in ~ if the language is that if the input and class ref.
;;;    Arguments:
;;;    input: some expression that must be a string
;;;    class-ref: a string naming a class
;;;    Return:
;;;    nil if not the case, something not nil otherwise.
;;;
;;; %PDM? (xx)                                                  [FUNCTION]
;;;    Checks if an object had the right PDM format, i.e. it must be a
;;;    symbol, bound, ~ be an alist, and have a local $TYPE property.
;;;    References are PDM objects
;;;
;;; %SUBTYPE? (object-id model-id &key (context *context*))     [FUNCTION]
;;;    Checks if object-id is a sub-type of model-id including itself.
;;;    Arguments:
;;;    object-id: identifier of object to check for modelhood
;;;    model-id: reference-model
;;;    context (opt): context default current
;;;    Return:
;;;    T or nil.
;;;
;;; %TYPE? (object-id class-id &key (context *context*))        [FUNCTION]
;;;    Checks if one of the classes of object-id is class-id or one of its
;;;    ~ subclasses.
;;;       If class is *any* then the result is true.
;;;    Arguments:
;;;    object-id: the id of a pdm object
;;;    class-id: the id of a class
;;;    Returns
;;;    nil or T.
;;;
;;; %TYPE-OF (obj-id &key (context *context*))                  [FUNCTION]
;;;    Extracts the type from an object (name of its class(es)).
;;;    Careful: returns "CONS" if following a programming error ~
;;;         obj-id pass the value of the identifier...
;;;    Argument:
;;;    obj-id: identifier of a particular object
;;;    Return:
;;;    a list containing the identifiers of the class(es) of which the
;;;    object is an instance ~ or its lisp type if it is not a PDM object.
;;;
```

8.3.3 Functions Dealing with the Internal Object Structure

```
;;; %%ALIVE? (obj-id context)                                   [FUNCTION]
;;;    Checks if object is PDM, *context* is OK, and object is alive. If
;;;    not, throws ~ to error.
;;;    Arguments:
;;;    obj-id: id
;;;    context (opt): context (default current)
```

```
;;;    Return:
;;;    resoved id, if OK, throws to :error otherwise.
;;;
;;; %ALIVE? (obj-id &key (context *context*))                        [FUNCTION]
;;;    Checks if object exists in given context ~
;;;         each object has a tombstone which records if it has been killed
;;;    ~ by somebody in the given context. This is a source of inconsistency
;;;    ~ since if the tombstone exists then the properties must be removed
;;;    in ~ current context. I.e. one cannot have a link onto an object
;;;    which ~ has been killed.
;;;    If context is illegal throws to :error.
;;;    Arguments:
;;;    obj-id: identifier of the object to be checked
;;;    context (opt): context (default current context)
;;;    Return:
;;;    nil if object is dead, resolved id otherwise
;;;
;;; %%ALLOWED-CONTEXT? (context)                                     [FUNCTION]
;;;    Checks if the context is allowed. Does this by looking at the global
;;;    variable *version-graph*. If not allowed throws to :error.
;;;
;;; %SP-GAMMA (node arc-label &key (context *context*))              [FUNCTION]
;;;    Returns the transitive closure ~
;;;         of objects for a given version.
;;;       This is the famous gamma function found in graph theory.
;;;       No check that arc-label is a structural property.
;;;    Arguments:
;;;    node: starting node
;;;    arc-label: label to follow on the arcs (e.g. $IS-A)
;;;    context (opt): context, default value is *context* (current)
;;;
;;; %SP-GAMMA-L (node-list arc-label &key (context *context*))       [FUNCTION]
;;;    Computes a transitive closure
;;;    Arguments:
;;;    node-list: a list of nodes from which to navigate
;;;    arc-label: the label of the arc we want to use
;;;    context (opt): context default current
;;;    Return:
;;;    a list of nodes including those of the node-list.
;;;
;;; %SP-GAMMA1 (ll prop stack old-set version)                       [FUNCTION]
;;;    Called by %sp-gamma for computing a transitive closure.
;;;       gamma1 does the job.
;;;       computes the whole stuff depth first using a stack of yet
;;;    unprocessed ~ candidates.
;;;     ll contains the list of nodes being processed
;;;     stack the ones that are waiting
;;;     old-set contains partial results.
;;;    Arguments:
;;;    ll: candidate list of nodes to be examined
```

```
;;;    prop: property along which we make the closure
;;;    stack: stack of waiting candidates
;;;    old-set: list containing the result
;;;    version: context
;;;    Return:
;;;    list of nodes making the transitive closure.
;;;
;;; %VG-GAMMA (context)                                             [FUNCTION]
;;;    Computes the transitive closure upwards on the version graph.
;;;       Get the transitive closure of the context i.e. a merge of all the
;;;    paths ~ from context to root without duplicates, traversed in a
;;;    breadth first ~ fashion, since we assume that the value should not be
;;;    far from our context ~ This list should be computed once for all when
;;;    the context is set ~ e.g. in a %set-context function
;;;    Argument:
;;;    context: starting context
;;;    Return:
;;;    a list of contexts.
;;;
;;; %VG-GAMMA-L (lres lnext ll pred)                                [FUNCTION]
;;;    Service function used be %vg-gamma
;;;    Arguments:
;;;    lres: resulting list
;;;    lnext: next candidates to examine later
;;;    ll: working list of candidates
;;;    pred: list of future nodes to process
;;;    Return:
;;;    lres
;;;
;;; %%ZAP-OBJECT (obj-id)                                           [FUNCTION]
;;;    Deletes an object by removing all values from all contexts and making
;;;    its ~ identifier unbound. This can destroy the database consistency.
;;;    Arguments:
;;;    obj-id: identifier of the object
;;;    Return:
;;;    not significant.
;;;
;;; %ZAP-PLIST (obj-id)                                             [FUNCTION]
;;;    Clears the plist of a given symbol.
;;;
```

8.3.4 Functions Dealing with Versions (Contest)

```
;;; %EXPLODE-OBJECT (obj-id &key into)                              [FUNCTION]
;;;    function that explodes an instance into a set of triples appending it
;;;    to the ~ list specified by the into variable.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    into (key): list to which to append new triples
;;;    Return:
```

```
;;;    the list of triples.
;;;
;;; INSERT-BEFORE (val val-l &rest val-test)                     [FUNCTION]
;;;    Should be a primitive.
;;;         Inserts item into a list either at the end of the list, or in
;;;    front of the ~ specified test-item, possibly duplicating it.
;;;    Arguments:
;;;    val: value to insert
;;;    val-l: list into which to insert the value
;;;    val-test (&rest): value that will be checked for insertion (equal
;;;    test) Return:
;;;       modified list or list of the value when val-l was nil
;;;
;;; INSERT-NTH (val val-l nth)                                   [FUNCTION]
;;;    Should be a primitive.
;;;         Inserts an item into a list at the nth position. If the list is
;;;    too short ~ then the value is inserted at the end of it.
;;;    Arguments:
;;;    val: value to be inserted
;;;    val-l: list into which to insert the value
;;;    nth: position at which the value must be inserted (0 is in front)
;;;    Return:
;;;    modified list
;;;
;;; %%MERGE-OBJECTS (obj-id obj-list context)                    [FUNCTION]
;;;    obj-id is the id of an object already existing in memory when we are
;;;    loading a new object with same id from disk. The new format of the
;;;    loaded object is obj-list. We compare properties of the existing
;;;    (moss) object and the loaded object and update all properties that
;;;    are not system or that contain application references. For a tp, the
;;;    system object is the reference. The result is a merged object
;;;    including system and application data. Arguments:
;;;    obj-id: id of the object to be processed
;;;    obj-list: object saved by the application
;;;    Returns:
;;;    the list representing the system object merged with application
;;;    references.
;;;
;;; %PUTC (obj-id value prop-id context)                         [FUNCTION]
;;;    Stores a versioned value onto the p-list of the object used as a
;;;    cache.
;;;
;;; %PUTM (object-id function-name method-name context           [FUNCTION]
;;;        &rest own-or-instance)
;;;    Records the function name corresponding to the method onto the ~
;;;         p-list of the object. Own-or-instance can be either one of the
;;;    ~ keywords :own or :instance; If not present, then :general is used.
;;;
;;; %%REMPROP (obj-id prop-id context)                           [FUNCTION]
;;;    Removes the values associated with ~
```

```
;;;         the specified property in the specified context. The net result
;;;    is ~ that the object inherits the values from a higher level.
;;;       So the resulting value in the context will probably not be nil.
;;;    Hence ~ it is not equivalent to a set-value with a nil argument.
;;;
;;; %%REMVAL (obj-id val prop-id context)                            [FUNCTION]
;;;    Removes the value associated with ~
;;;         a property in a given context. Normally useful for terminal
;;;    properties ~ However does not check for entry-points. Also val must
;;;    be a normalized value n i.e. must be expressed using the internal
;;;    format (it cannot be an external value).
;;;
;;; %%REMNTHVAL (obj-id val prop-id context position)                [FUNCTION]
;;;    Removes the nth value associated ~
;;;         with a property in a given context. Normally useful for
;;;    terminal properties ~ However does not check for entry-points. Does
;;;    not return any significant value.
;;;
;;; %RESOLVE (id &key (context *context*))                           [FUNCTION]
;;;    Replaces id with the actual id when it is a ref object. Otherwise
;;;    returns id. When id is unbound returns id.
;;;    Arguments:
;;;    id: object id or REF identifier
;;;    Return:
;;;    resolved id.
;;;
;;; %%SET-SYMBOL-PACKAGE (symbol package)                            [FUNCTION]
;;;    if package is nil set it to default *package*, and create symbol in
;;;    the ~ specified package. No check on input.
;;;    Arguments:
;;;    symbol: any symbol
;;;    package: a valid package or keyword
;;;    Return:
;;;    a symbol interned in the package.
;;;
;;; %%SET-VALUE (obj-id value prop-id context)                       [FUNCTION]
;;;    Resets the value associated ~
;;;         with a property in a given context. Replaces previous values.
;;;    Does not ~ try to inherit using the version-graph, but defines a new
;;;    context locally ~ Thus, it is not an %add-value, but an actual
;;;    %%set-value.
;;;
;;; %%SET-VALUE-LIST (obj-id value-list prop-id context)             [FUNCTION]
;;;    Resets the value associated ~
;;;         with a property in a given context. Replaces previous values.
;;;    Does not ~ try to inherit using the version-graph, but defines a new
;;;    context locally ~ Thus, it is not an %add-value, but an actual
;;;    %%set-value - ~ No entry point is created.
;;;
```

8.3.5 Functions Dealing with Properties

Attributes

```
;;; %%ADD-VALUE (obj-id prop-id value context)                    [FUNCTION]
;;;    Inserts a new value for prop-id, no checking done on arguments
;;;    Context must be ~ explicitly given.
;;;    Arguments:
;;;    obj-id: id of object
;;;    prop-id: id of property for which we add the value
;;;    value: value to add
;;;    context: context in which we add the value
;;;    Return:
;;;    the list representing the modified object.
;;;
;;; %ADD-VALUE (obj-id prop-id value &key (context *context*)        [FUNCTION]
;;;              before-value allow-duplicates)
;;;    Inserts a new value for a given property in the specified context. ~
;;;         Copies the old values from a previous context if necessary.
;;;    We assume that prop-id is the correct local property, and that ~
;;;         the value has been validated.
;;;    Arguments:
;;;    obj-id: id of the object to be modified
;;;    prop-id: id of the target property
;;;    value: value to be added
;;;    context: context in which the value is added
;;;    before-value (key): value - specifies that we want to insert in
;;;    front of value allow-duplicates (key): t - allows to insert duplicate
;;;    values. No cardinality checking done on args. If value is there does
;;;    not do ~ anything.
;;;    Throws to an :error tag if the context is not allowed
;;;    Return:
;;;    the internal format of the modified object
;;;
;;; %VALIDATE-TP (tp-id value-list &key (context *context*))        [FUNCTION]
;;;    Checks if the property values obey the restrictions attached to the
;;;    attribute. Arguments:
;;;    tp-id: id of the attribute
;;;    value-list: list of values to check
;;;    context (opt): context (default current)
;;;    Return:
;;;    2 values:
;;;          1st value value-list if OK, nil otherwise
;;;          2nd value: list of string error message.
;;;
;;; %VALIDATE-TP-VALUE (restriction data &key prop-id obj-id        [FUNCTION]
;;;                      (context *context*))
;;;    Takes some data and checks if they agree with a value-restriction
;;;    condition,~ associated with a terminal property. Used by the =xi
;;;    method whenever ~ we want to put data as the value of the property.
;;;    Processed conditions are those of *allowed-restriction-operators*
```

```
;;;    Arguments:
;;;    restriction: restriction condition associated with the property,
;;;    e.g. (:one-of (1 2 3)) data: a list of values to be checked
;;;    prop-id (key): identifier of the local property (unused)
;;;     obj-id (key): identifier of the object being processed (unued)
;;;    context (key): context default current (unused)
;;;    Return:
;;;    data when validated, nil otherwise.
;;;
```

Relations

```
;;; %LINK (obj1-id sp-id obj2-id &key (context *context*))           [FUNCTION]
;;;    Brute force link two objects using the structural property id - ~
;;;         No cardinality check is done - sp-id must be direct property.
;;;       (%link obj1-id sp-id obj2-id context) - links two objects ~
;;;         using sp-id - no checking done on args - crude link to be able
;;;    ~ to bootstrap without the properties really existing. Here we
;;;    manipulate ~ the names. We assume that sp-id is not an inverse
;;;    property ~ If the link already exists then does nothing. Otherwise,
;;;    introduces a ~ new link modifying or creating a value in the
;;;    specified context. ~ This might look like a bit complicated a piece
;;;    of code, however we try ~ to do simple tests first to avoid long
;;;    execution times. Arguments:
;;;    obj1-id: identifier for first object
;;;    sp-id: identifier of the local linking property
;;;    obj2-id: identifier for the second object
;;;    context: context
;;;    Return:
;;;    list representing the first object
;;;
;;; %LINK-ALL (obj1-id sp-id obj-list &key (context *context*))      [FUNCTION]
;;;    Brute force link an object to a list of objects using the structural
;;;    property ~ id - No cardinality check is done - sp-id must be direct
;;;    property. (%link obj1-id sp-id obj-list context) - links two objects
;;;    ~ using sp-id - no checking done on args - crude link to be able ~
;;;         to bootstrap without the properties really existing. Here we
;;;    manipulate ~ the names. We assume that sp-id is not an inverse
;;;    property ~ If the link already exists then does nothing. Otherwise,
;;;    introduces a ~ new link modifying or creating a value in the
;;;    specified context. ~ This might look like a bit complicated a piece
;;;    of code, however we try ~ to do simple tests first to avoid long
;;;    execution times. Arguments:
;;;    obj1-id: identifier for first object
;;;    sp-id: identifier of the local linking property
;;;    obj-list: identifier for the second object
;;;    context: context
;;;    Return:
;;;    list representing the first object
;;;
```

```
;;; %UNLINK (obj1-id sp-id obj2-id &key (context *context*))        [FUNCTION]
;;;    Disconnects to objects by removing the structural link.
;;;         in the given context. If the context does not exist then one
;;;    has to ~ recover the states of the links in the required context.
;;;    Then if the ~ link did not exist, then one exits doing nothing.
;;;    Otherwise, one removes ~ the link and stores the result.
;;;
;;; %VALIDATE-SP (sp-id suc-list &key (context *context*))           [FUNCTION]
;;;    Checks if the property successors obey the restrictions attached to
;;;    the relation. Arguments:
;;;    sp-id: id of the relation
;;;    suc-list: list of successors to check
;;;    context (opt): context (default current)
;;;    Return:
;;;    2 values:
;;;          1st value value-list if OK, nil otherwise
;;;          2nd value: list of string error message.
;;;
```

Inverse Relations

8.3.6 Constructors

```
;;; %BUILD-METHOD (function-name arg-list body)                      [FUNCTION]
;;;    build a MOSS method from its definition in the various defxxxmethof
;;;    macros. ~ Body may have a comment (string) and/or a declare
;;;    statement. The method ~ body is enclosed in a catch :return clause to
;;;    allow exits, since we cannot ~ use the return-from clause inside the
;;;    method. When the compiler is included in the environment, compiles
;;;    the code. Arguments:
;;;    function-name: the name of the function implementing the method
;;;    (e.g. $UNI=I=0=SUMMARY) arg-list: method arguments
;;;    body: the body of the function
;;;    Return:
;;;    the function name.
;;;
;;; %CREATE-BASIC-NEW-OBJECT (object-id &key (context *context*) id  [FUNCTION]
;;;                           ideal (package *package*))
;;;    Creates a skeleton of object containing only the $TYPE and $ID ~
;;;         properties in the current context.
;;;    Arguments:
;;;    object-id: presumably class of the object to be created
;;;    context (key): context default current
;;;    id (key): if there, id is specified (we do not create it)
;;;    ideal (key): if t, we want to create an ideal instance
;;;                        (with sequence number 0)
;;;    package (key): package (default is current)
;;;    Return:
;;;    id of the newly created object.
;;;
;;; %CREATE-BASIC-ORPHAN (&key (context *context*)                   [FUNCTION]
```

```
;;;                          (package *package*))
;;;    Creates a skeleton for an orphan object and the associated key.
;;;    Arguments:
;;;    context (key): context (default current)
;;;    Return:
;;;    the id of the new orphan.
;;;
;;; %CREATE-ENTRY-POINT-METHOD (object-id object-name               [FUNCTION]
;;;                                 &optional args doc body)
;;;    Creates an entry point method, with default if no arguments. ~
;;;         the =make-entry symbol is defined in the same package as
;;;    object-id. If args is present it must contain an option &key (package
;;;    *package*). Arguments:
;;;    object-id: id to which the =make-entry method will be attached
;;;    object-name: name of the object (for error message)
;;;    args (opt): arguments of the method (default is (value-list))
;;;    doc (opt): documentation string
;;;    body (opt): body of the method
;;;    Return:
;;;    an own-method id.
;;;
;;; %INVERSE-PROPERTY-ID (prop-id &key (context *context*))          [FUNCTION]
;;;    Computes the inverse property id of a given structural or terminal
;;;    property Optional arg: version. When the property is still undefined,
;;;    calls %make-id-for-inverse-property.
;;;    For inverse properties of an inverse property return a list if
;;;    more than one, ~ e.g. when we have a property lattice.
;;;    Arguments:
;;;    prop-id: identifier of property
;;;    context (opt): context default current
;;;    Returns:
;;;    an inverse property identifier
;;;
;;; %MAKE-ENTITY-SUBTREE (ens &key (context *context*))              [FUNCTION]
;;;    takes an entity and builds the subtree of its sub-classes, e.g.
;;;    (A (B C (D E) F))
;;;    where (D E) are children of C and (B C F) are children of A.
;;;    Arguments:
;;;    ens: entity to process
;;;    Return:
;;;    (ens) if no subtree, or (ens <subtree>).
;;;
;;; %MAKE-ENTITY-TREE (&key (context *context*))                     [FUNCTION]
;;;    builds the forest corresponfing to the application classes.
;;;    Arguments:
;;;    context (opt): default is *context*
;;;    Return:
;;;    a tree like (A (B C (D E) F) G H)
;;;    where (D E) are children of C and (B C F) are children of A, and G
;;;    and H are parallel trees.
```

```
;;;
;;; %MAKE-ENTITY-TREE (&key (context *context*)                         [FUNCTION]
;;;                       (package *package*))
;;;    builds the forest keeping classes from the current package.
;;;    Arguments:
;;;    context (opt): default is *context*
;;;    Return:
;;;    a tree like (A (B C (D E) F) G H)
;;;    where (D E) are children of C and (B C F) are children of A, and G
;;;    and H are parallel trees.
;;;
;;; %MAKE-ENTRY-SYMBOLS (multilingual-name &key type prefix             [FUNCTION]
;;;                         (package *package*))
;;;    Takes a multilingual-name as input and builds entry symbols for each
;;;    of the ~ synonyms in each specified language.
;;;    Note that a list of symbols is not a valid input argument.
;;;    Arguments:
;;;    multilingual-name: string, symbol or multilingual-name specifying
;;;    the entry points tp-id: local attibute identifier
;;;    type (key): type of MOSS object (default is nil)
;;;    prefix (key): prefix for produced symbols
;;;    Return:
;;;    list of the entry point symbols
;;;
;;; %MAKE-EP (entry tp-id obj-id &key (context *context*) export)    [FUNCTION]
;;;    Adds a new object to an entry-point if it exists - Otherwise creates
;;;    the entry point(s) and record it/them in the current system
;;;    (*moss-system*). Export the entry symbol(s).
;;;    Arguments:
;;;    entry: symbol or list of symbols specifying the entry point(s)
;;;    tp-id: local attibute identifier
;;;    obj-id: identifier of object to index
;;;    context (opt): context (default current)
;;;    export (key): it t export entry points from the package
;;;    Return:
;;;    internal format of the entry point object or a list of them
;;;
;;; %MAKE-EP-FROM-MLN (mln tp-id obj-id &key type                       [FUNCTION]
;;;                       (context *context*) export)
;;;    Takes a multilingual-name as input and builds entries for each of the
;;;    ~ synonyms in each specified LEGAL language in package of obj-id.
;;;    Arguments:
;;;    mln: legal multilingual-name specifying the entry points
;;;    tp-id: local attibute identifier
;;;    obj-id: identifier of object to index
;;;    context (opt): context (default current)
;;;    type (key): type of object, e.g. $ENT, $EPR, $FN, ... (see
;;;    %make-name) export (key): if t then we should export entry points
;;;    from their package Return:
;;;    list of the entry point object (internal format)
```

```
;;;
;;; %MAKE-ID (class-id &key name prefix value context id ideal      [FUNCTION]
;;;           (package *package*))
;;;    Generic function to make object identifiers. If the symbol exists,
;;;    then ~ we throw to :error.
;;;    Arguments:
;;;    class-id: identifier of the class of the object (*none* is
;;;    authorized) name (key): useful name, e.g.class  name or property name
;;;    prefix (key): additional name (string), e.g. for defining
;;;    properties (the class name) value (key): value for an instance
;;;    context (key): context default current
;;;    id (key): id of class or property
;;;    ideal (key): if there indicates we want an ideal, :id option must
;;;    be there package (key): package into which new symbol must be
;;;    inserted (default *package*) Return:
;;;    unique new symbol.
;;;
;;; %MAKE-INVERSE-PROPERTY (id multilingual-name                    [FUNCTION]
;;;                         &key (context *context*) export)
;;;    create an inverse property for an attribute or a relation. Links it
;;;    to the ~ direct property, creates the name and entry point, add it to
;;;    MOSS. It is created in the same package as id and in the specified
;;;    context. Uses the default language.
;;;    Arguments:
;;;    id: identifier of the property to invert
;;;    multilingual-name: name of the property to invert
;;;    Return:
;;;    :done
;;;
;;; %MAKE-INVERSE-PROPERTY-MLN (mln)                                [FUNCTION]
;;;    build an inverse multilingual name for the inverse property.
;;;    Arguments:
;;;    mln: multilingual name
;;;    Result:
;;;    a list of strings and an inverse mln
;;;
;;; %MAKE-NAME (class-id &rest option-list &key name                [FUNCTION]
;;;             (package *package*) &allow-other-keys)
;;;    Makes a name with %make-name-string and interns is in the specified
;;;    package. Arguments:
;;;    class-id: identifier of the corresponding class
;;;    package (key): package for interning symbol (default current
;;;    execution package) name (key): string, symbol or multilingual string
;;;     more keys are allowed as specified in the lambda-list of the
;;;    %make-string function Return:
;;;    interned symbol.
;;;
;;; %MAKE-NAME-STRING (class-id &key name method-class-id           [FUNCTION]
;;;                    method-type state-type short-name prefix
;;;                    (type "") (context *context*)
```

```
;;;                   &allow-other-keys &aux pfx)
;;;    Builds name strings for various moss entities, e.g. method names,
;;;    inverse-property ~ names, internal method function names, etc.
;;;    Examples of resulting strings
;;;        XXX-Y-ZZZ from " Xxx y'zzz "
;;;        IS-XXX-OF inverse property
;;;        HAS-XXX  property
;;;        $E-PERSON=S=0=PRINT-SELF own method internal function name
;;;        $E-PERSON=I=0=SUMMARY instance method internal function name
;;;       *0=PRINT-OBJECT universal method internal function name
;;;        $-PER typeless radix
;;;        _HAS-BROTHER  internal variable
;;;    We mix English prefix and suffix with other languages, which is
;;;    not crucial ~ since such names are internal to MOSS.
;;;    Arguments:
;;;    class-id: identifier of the corresponding class
;;;    name (key): name, e.g. of method
;;;    context (key): context reference
;;;    prefix (key): symbol, string or mln specifying a class (for class
;;;    properties) method-type (key): instance or own or universal
;;;    method-class-id (key): class-id for a given method
;;;    state-type: for state objects {:entry-state, :success, :failure}
;;;    short-name: for state objects, prefix
;;;    package (key): package for interning symbol (default current
;;;    execution package) context (key): context used for building some
;;;    names (default current) language (key): language to use, e.g. :en,
;;;    :fr (default *language*) Return:
;;;    a single name string
;;;
;;; %MAKE-PHRASE (&rest words)                                       [FUNCTION]
;;;    Takes a list of words and returns a string with the words separated
;;;    by a space. Arguments:
;;;    words (rest): strings
;;;    Return:
;;;    a string.
;;;
;;; %MAKE-RADIX-FROM-NAME (name &key (type "") (package *package*))  [FUNCTION]
;;;    build a radix name for a property or for a class. Tries to build a
;;;    unique name ~ using the first 3 letters of the name. Error if the
;;;    symbol already exists. Arguments:
;;;    name: e.g. "SIGLE"
;;;    type (opt): e.g. "T " (for terminal property)
;;;    Return:
;;;    e.g. $T-SIG or error if already exists
;;;
;;; %MAKE-REF-OBJECT (class-ref obj-id &key (context *context*))     [FUNCTION]
;;;    Builds a reference object, checking the validity of the class. If
;;;    invalid ~ throws to :error tag.
;;;    Arguments:
;;;    class-ref: reference of the new class
```

```
;;;    obj-id: id of object being referenced
;;;    context (opt): context, default current
;;;    Return:
;;;    id of new ref object.
;;;
;;; %MAKE-STRING-FROM-PATTERN (control-string pattern                [FUNCTION]
;;;                            &rest properties)
;;;    makes a string from a pattern (XML like). The first element of
;;;    pattern is the ~ class of the  object. The function ~
;;;    is somewhat equivalent to the =summary method, but the method is
;;;    applied to ~ the structured object.
;;;    Ex: (fn "~{~A~^ ~}, ~{~A~^ ~}"
;;;              '("person" ("name" "Dupond") ("first-name" "Jean"))
;;;              "name" "first-name")
;;;    returns "Dupond, Jean"
;;;    Arguments:
;;;    pattern: the pattern
;;;    data: the data
;;;    Return:
;;;    string to be printed.
;;;
;;; %MAKE-WORD-COMBINATIONS (word-list)                              [FUNCTION]
;;;
;;; %MAKE-WORD-LIST (text &key (norm t))                             [FUNCTION]
;;;    Norms a text string by separating each word making it lower case with
;;;    a leading~ uppercased letter.
;;;    Arguments:
;;;    text: text string
;;;    norm (key): if true (default) capitalize the first letter of each
;;;    word Return:
;;;    a list of normed words.
;;;
```

8.3.7 Extraction Functions

```
;;; %ACCESS-ENTITIES-FROM-WORD-LIST (word-list                       [FUNCTION]
;;;                                  &rest empty-word-lists)
;;;    takes a list of words, removes the empty words and tries to find
;;;    entities ~ of the KB using entry points from the list of words and
;;;    filtering the results ~ to keep only best matches.
;;;    Arguments:
;;;    word-list: input list of words
;;;    empty-word-list (key): list of empty words to remove from word
;;;    list Return:
;;;    list of object ids or nil.
;;;
;;; %EXTRACT (entry tp class &key (context *context*)                [FUNCTION]
;;;           (filter nil filter-there?)
;;;           (class-ref nil class-there?)
;;;           (package *package* package-there?)
```

```
;;;           allow-multiple-values)
;;;    Extracts an object that is specified by its entry-point - attribute -
;;;    ~ concept - and context. When more than one, applies a function when
;;;    ~ specified in the option <:filter function>. If more than one value
;;;    left ~ then an error is declared, unless they are explicitely allowed
;;;    by setting the ~ keyword argument :allow-multiple-values to T.
;;;    Arguments:
;;;    entry: entry point (string, symbol or mln)
;;;    tp: attribute (string, symbol or mln)
;;;    class: class name (string, symbol or mln)
;;;    context (key): context default current
;;;    filter (key): filter function when more than one value is expected
;;;    filter-there?: variable set to T when a filter is specified
;;;    class-ref (key): reference of class to locate a property
;;;    class-there?: variable set to T when class-ref is specified
;;;    package (key): default current
;;;    allow-multiple-values (key): if T the function may return more
;;;    than one value Returns
;;;    nil if it cannot find anything
;;;    a symbol  for a single object
;;;    a list when it finds several objects and multiples are allowed
;;;
;;; %EXTRACT-FROM-ID (entry prop-id class-id                         [FUNCTION]
;;;                 &key (context *context*))
;;;    Returns the list of all objects  corresponding to entry point entry ~
;;;         with inverse attribute of prop-id and of class class-id.
;;;       E.g. (%extract-from-id 'ATTRIBUTE '$ENAM '$ENT 0) returns ($EPT)
;;;       obj-id are returned if they are in class class-id or in one of
;;;    the subclasses ~ (transitive closure along the $IS-A.OF link).
;;;       Also all meta-models $ENT, $EPS, $EPT, $FN, $CTR ($EIL?) are
;;;    defined ~ as instances of $ENT.
;;;       If context is illegal throws to an :error tag.
;;;       The function works with the *any* and *none* pseudo classes:
;;;     (%extract-from-id <entry> <prop-id> '*any*) returns everything
;;;     (%extract-from-id <entry> <prop-id> '*none*) returns only classless
;;;    objects. We also allow objects with multiple classes.
;;;    No checks are done to speed execution.
;;;    Arguments:
;;;    entry: entry point (symbol)
;;;    prop-id: id of property to be used to make the inverse
;;;                 should be the exact property associated with class-id
;;;    class-id: id of the potential class of the object
;;;    context (key): context (default current)
;;;    Returns:
;;;    a list of objects maybe one
;;;
;;; %GET-ALL-ATTRIBUTES "()"                                        [FUNCTION]
;;;    Gets the list of all attributes in all contexts. No arguments.
;;;
;;; %%GET-ALL-CLASS-ATTRIBUTES (class-id &key (context *context*))   [FUNCTION]
```

```
;;;    Collects all attributes starting with class-id through superclasses.
;;;    Attributes of the same property tree can be included in the
;;;    result. Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-INVERSE-RELATIONS (class-id                      [FUNCTION]
;;;                                    &key (context *context*))
;;;    Collects all relations starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-PROPERTIES (class-id &key (context *context*))   [FUNCTION]
;;;    Collects all properties starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-RELATIONS (class-id &key (context *context*))    [FUNCTION]
;;;    Collects all relations starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %GET-ALL-CONCEPTS "()"                                           [FUNCTION]
;;;    Gets the list of all concepts in all contexts. No arguments.
;;;
;;; %GET-ALL-ENTRY-POINTS "()"                                       [FUNCTION]
;;;    Gets the list of all entry-points in all contexts. No arguments.
;;;
;;; %GET-ALL-INSTANCES (class-id &key package)                       [FUNCTION]
;;;    Gets the list of all instances of a class in all contexts. We assume
;;;    that ~ counter is in context 0.
;;;    The default behavior is that the package for the instances is the
;;;    same as ~ the package of the class-id symbol. Otherwise one should
;;;    know what ~ one does...
;;;    Arguments:
;;;    class-id: id of the class
```

```
;;;    return:
;;;    list of all bound identifiers of the instances of the class
;;;
;;; %GET-ALL-INVERSE-RELATIONS "()"                             [FUNCTION]
;;;    Gets the list of all inverse relations in all contexts. No arguments.
;;;
;;; %GET-ALL-METHODS "()"                                       [FUNCTION]
;;;    Gets the list of all methods in all contexts. No arguments.
;;;
;;; %GET-ALL-MOSS-OBJECTS "()"                                  [FUNCTION]
;;;    Gets the list of all MOSS objects in all contexts. No arguments.
;;;
;;; %GET-ALL-ORPHANS "()"                                       [FUNCTION]
;;;    Gets the list of all orphans in all contexts. No arguments.
;;;
;;; %GET-ALL-RELATIONS "()"                                     [FUNCTION]
;;;    Gets the list of all relations in all contexts. No arguments.
;;;
;;; %%GET-ALL-SUBCLASS-INVERSE-RELATIONS (class-id              [FUNCTION]
;;;                                      &key (context *context*))
;;;    Collects all properties starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-SUBCLASS-RELATIONS (class-id                      [FUNCTION]
;;;                              &key (context *context*))
;;;    Collects all relations starting with class-id through subclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %GET-ALL-SYMBOLS "()"                                       [FUNCTION]
;;;    Gets the list of all symbols in all contexts (counters, global
;;;    variables,... No arguments.
;;;
;;; %GET-ALL-VERSIONS (obj-id prop-id)                          [FUNCTION]
;;;    Returns all versions of values or of successors associated with ~
;;;         the given property prop-id. Removes ~
;;;         duplicates of the values corresponding to the various versions.
;;;    ~ The function is used when trying to get all objects of a kind ~
;;;         from the system, e.g. by %get-all-symbols.
;;;    Arguments:
;;;    obj-id: object identifier
```

```
;;;    prop-id: property id for which we want all values
;;;    Return:
;;;    a list of all values regardless of versions.
;;;
;;; %GET-AND-INCREMENT-COUNTER (obj-id                               [FUNCTION]
;;;                            &optional (counter-link-id '$ctrs))
;;;    Obtains the value of a counter associated with obj-id and increases
;;;    this value ~ in the counter itself. Throws to :error if obj-id is
;;;    invalid. Arguments
;;;    obj-id: id of the object that owns the counter (ususally a class)
;;;    counter-link-id (opt): name of the property linking to the counter
;;;    (default $CTRS) Return:
;;;    old value of the counter.
;;;
;;; %GET-APPLICATION-CLASSES (&key (context *context*))              [FUNCTION]
;;;    returns the list of classes of a given application by extracting them
;;;    from ~ the system object and removing the system classes.
;;;    If OMAS is active, then we consider objects from the current
;;;    package. Arguments:
;;;    context (opt): default is *context*
;;;    Returns:
;;;    a list of class ids.
;;;
;;; %GETC (obj-id prop-id context)                                   [FUNCTION]
;;;    Retrieves the value associated with prop-id ~
;;;         for the specified context from the p-list of obj-id used as a
;;;    cache. ~ If the context is illegal, then throws to an :error tag.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of local property
;;;    context: context
;;;    Returns:
;;;    nil or the cached value, Throws to :error when context is illegal
;;;    or object dead.
;;;
;;; %GET-CLASS-CANONICAL-NAME-FROM-REF (ref)                         [FUNCTION]
;;;    uses ref as an entry point to the property. If it fails return nil.
;;;    Otherwise, ~ returns the canonical name in the current language.
;;;    Arguments:
;;;    ref: e.g. "country"
;;;    Return:
;;;    canonical name in the current language specified by *language*.
;;;
;;; %GET-CLASS-ID-FROM-REF (class-ref &key (context *context*)       [FUNCTION]
;;;                        (package *package*))
;;;    Retrieves the class id from its name. Should be unique. Error
;;;    otherwise. Argument:
;;;    class-ref: a symbol, string or multilingual name, e.g. PERSON,
;;;    "Person", (:en ...) context (opt): context default current
;;;    package (opt): default current
```

```
;;;    Return:
;;;    class-id, e.g. $E-PERSON
;;;
;;; GET-CURRENT-YEAR "()"                                          [FUNCTION]
;;;
;;; %GET-DEFAULT-FROM-CLASS (prop-id class-id)                     [FUNCTION]
;;;    gets the default value-list associated with a property in a given
;;;    class. ~ The way to do it is to ask the property itself; then the
;;;    ideal of the class. Defaults cannot be inherited.
;;;    Arguments:
;;;    prop-id: id of corresponding property
;;;    class-id: id of the class
;;;    Return:
;;;    defaul value or nil.
;;;
;;; %GET-OBJECT-CLASS-NAME (obj-id)                                [FUNCTION]
;;;    get the class names of an object.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    Return:
;;;    a list of strings representing the names of the classes to which
;;;    the object belongs.
;;;
;;; %GET-EMPTY-WORDS (language)                                    [FUNCTION]
;;;    recover a list of empty words in specified language. This assumes ~
;;;             that a class empty-words exists and has been initialized.
;;;    Arguments:
;;;    language: a string or keyword
;;;    Return:
;;;    a list of empty words or nil.
;;;
;;; %GET-GENERIC-PROPERTY (prop-list &key (context *context*))     [FUNCTION]
;;;    We start with a set of properties having the same name and try to
;;;    obtain the ~ one that is generic, i.e. that does not have an ancestor
;;;    (no $IS-A property). Argument:
;;;    prop-list: list of property identifiers
;;;    Return:
;;;    the identifier of the generic property or nil. Warns if there are
;;;    more than one left.
;;;
;;; %GET-GENERIC-PROPERTY-FROM-NAME (prop-name                     [FUNCTION]
;;;                                  &key (context *context*))
;;;    We start with a property name and try to obtain the id of the ~
;;;         one that is generic, i.e. that does not have an ancestor.
;;;    Argument:
;;;    prop-name: property name
;;;    Return:
;;;    the identifier of the generic property or nil. Warns if there are
;;;    more than one left.
;;;
```

```
;;; %GET-GENERIC-PROPERTY-FROM-REF (prop-ref)                       [FUNCTION]
;;;    We start with a set of properties having the same name and try to
;;;    obtain the ~ one that is generic, i.e. that does not have an ancestor
;;;    (no $IS-A property). Argument:
;;;    prop-ref: string or symbol e.g. "content" or CONTENT
;;;    Return:
;;;    the identifier of the generic property (e.g. HAS-CONTENT) or nil.
;;;
;;; %GET-INDEX-WEIGHTS (task-id)                                    [FUNCTION]
;;;    get a list of pairs index (a string) value from the list of ~
;;;        index patterns associated with the task.
;;;    Arguments:
;;;    task: a task object
;;;    Return:
;;;    a list of pairs.
;;;
;;; %GET-INSTANCE-COUNT (class-id &key (context *context*)          [FUNCTION]
;;;                        (allow-subclasses t))
;;;    Gets an upper-bound on the number of instances for a given class.
;;;    Arguments:
;;;    class-id: identifier of class
;;;    allow-subclasses (key): if T counts instances of subclasses
;;;    Returns:
;;;    a number
;;;
;;; %%GET-LIST-OF-VALUES-AND-CONTEXT (obj-id prop-id)               [FUNCTION]
;;;    Done to obtain raw value list of objects.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: property local identifier
;;;    Returns:
;;;    a list of all values by context (internal format of the set of
;;;    values).
;;;
;;; %GETM (object-id method-name context &rest own-or-instance)     [FUNCTION]
;;;    When methods are compiled on p-list it is done as follows:
;;;     (=print-self (:own (nn <int-function name>)*) (:instance  <...>))
;;;       if the method type is not specified then the format is slightly ~
;;;         different from the more general format.
;;;       The version must be recorded locally otherwise we shall try ~
;;;         to inherit it. Remember this is a cache, not an object.
;;;       When using versions we can have the method cached onto the p-list
;;;    for a ~ previous version. If the method has not been modified since,
;;;    it is a waste ~ of energy to recover it again, to recompile it, and
;;;    to give it a new name. ~ Rather, we want to check that it was not
;;;    modified along the same branch indeed, ~ then we cache it under the
;;;    current version explicitely with the old internal ~ name. In order to
;;;    do that, we must record every time a method code or arguments ~ was
;;;    changed. Doing
;;;           (%get-value <method-id> <modif> *current-context*)
```

```
;;;       will return the last modification or nil (it could have been
;;;    modified, but ~ then it is not in the same branch.
;;;       So for recovering a cached method, we first get the current
;;;    branch of the ~ path from the current context to the root of the
;;;    version graph. We then loop on the successive values of the versions
;;;    in the branch starting ~ with the current one. At each step, either
;;;    we get a function, or this is the ~ step at which the version was
;;;    modified, in which case we loose and we must ~ recompute and
;;;    recompile the method. Otherwise, when we get a previously ~ recorded
;;;    method, we must cache it explicitely on the p-list for next time
;;;    around. Arguments: object-id: id of the object
;;;    method-name: name of the target method
;;;    context: context
;;;    own-or-instance (key): flag to specify type of method
;;;    Return:
;;;    method or nil
;;;
;;; %GET-NAME-FROM-REF (ref object-type)                          [FUNCTION]
;;;    Takes a reference and tries to obtain the official name for the
;;;    object, e.g. ~ "class" -> CONCEPT
;;;    If ref is a symbol, we assume it is the name we are looking for,
;;;       if ref is a multilingual name, we use the canonical name,
;;;       if ref is a string, we build a name from it, specific to the
;;;    object type, ~ and recover the canonical name from its recorded name.
;;;    if the object is a property yet undefined, we return the cooked up
;;;    name. Arguments:
;;;    ref: a symbol, string or multilingual name
;;;    object-type: a keyward giving the type of the object (:class,
;;;    :attribute, ...) package (opt): a package spec for the name
;;;    Return:
;;;    the canonical name of the object.
;;;
;;; %%GET-OBJECTS-FROM-ENTRY-POINT (entry &key (context *context*)   [FUNCTION]
;;;                                 keep-entry)
;;;    Sets up an a-list (inv-prop obj-id) of all objects for which entry is
;;;    an ~ entry point, eliminating the current-system (inv prop:$EPLS.OF).
;;;    Arguments:
;;;    entry: entry-point
;;;    context (key): context default current
;;;    keep-entry (key): if t keep the entry point
;;;    Return:
;;;    a list of pairs (<inv-prop-id> . <obj-id>)  or nil
;;;
;;; %GET-PROP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref        [FUNCTION]
;;;                                       &key (context *context*)
;;;                                       (package *package*))
;;;    Determines the right property for the specific class. Throws to
;;;    :error when ~ something is wrong. Property may be inherited.
;;;    Arguments:
;;;    prop-name: name of a property
```

```
;;;    class-ref: class name or class identifier
;;;    context (opt): context default current
;;;    package (opt): defualt current
;;;    Return:
;;;    the property attached to the class.
;;;
;;; %GET-PROPERTIES (object-id &key (context *context*))            [FUNCTION]
;;;    Obtain all the possible properties for a given object. If an orphan,
;;;    then ~ we only get local properties attached to the orphan.
;;;    Otherwise, we get ~ all inherited properties corresponding to the
;;;    object class. Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of properties associated with the object.
;;;
;;; %GET-PROPERTY-CANONICAL-NAME-FROM-REF (ref)                     [FUNCTION]
;;;    uses ref as an entry point to the property. If it fails return nil.
;;;    Otherwise, ~ returns the canonical name in the current language.
;;;    Arguments:
;;;    ref: e.g. "country"
;;;    Return:
;;;    canonical name in the current language specified by *language*.
;;;
;;; %GET-PROPERTY-ID-FROM-NAME (obj-id prop-ref                     [FUNCTION]
;;;                             &key (context *context*)
;;;                             (package *package*))
;;;    Function that has the difficult role of finding the right property
;;;    that ~ applies to the object, and corresponds to the prop-name. This
;;;    is ~ difficult in case we have tree-properties.
;;;    It is used by elementary universal methods like =get =put =delete
;;;    Result is cached onto the p-list of the property name (e.g.
;;;    HAS-NAME). Arguments:
;;;    obj-id: object for which we try to determine property
;;;    prop-ref: property ref, e.g. HAS-FIRST-NAME, "first name", or mln
;;;    context (opt): context default current
;;;    Return:
;;;    the identifier of the corresponding property or nil if the class
;;;    inherits property.
;;;
;;; %GET-PROPERTY-SYNONYMS (prop-ref &key (package *package*))      [FUNCTION]
;;;    takes a property reference and return the list of synonyms for this
;;;    property. Arguments:
;;;    prop-ref: a property reference, e.g. "street"
;;;    package (key): package in which property name has been defined
;;;    Return;
;;;    nil or the list of all property synonyms.
;;;
;;; %GET-RELEVANT-WEIGHTS (weight-list word-patterns                [FUNCTION]
;;;                        &key used-patterns)
```

```
;;;    function to obtain weigths for a list of words.
;;;    Arguments:
;;;    weight-list: list of weights e.g. ((<string> .6))
;;;    word-patterns: the list of expressions (strings) to test
;;;    used-patterns (key): expressions that have already been tried
;;;    Return:
;;;    a list of pairs string weight.
;;;
;;; %GET-SP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref          [FUNCTION]
;;;                                    &key (context *context*)
;;;                                    (package *package*))
;;;    Determines the right relation for the specific class. Throws to
;;;    :error when ~ something is wrong.
;;;    Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    the property attached to the class.
;;;
;;; %GET-SP-VALUE-RESTRICTIONS (prop-id &key (context *context*))    [FUNCTION]
;;;    Obtains the value restrictions from the attribute representation. No
;;;    ~ inheritance is done.
;;;    Restrictions are
;;;       :exists     ; among the possible values one must be of that type
;;;       :forall     ; all values should be of that type
;;;       :type       ; values should be of that type
;;;       :not-type   ; no value should be of that type
;;;       :value      ; only this value is allowed
;;;       :one-of     ; the value should be a member of this list
;;;       :not        ; the value should not be one of this list
;;;    Arguments:
;;;    prop-id: attribute id
;;;    context (opt): context (default current)
;;;    Returns:
;;;    a list of restrictions: e.g. ((:one-of 1 2 3) (:type :integer)
;;;    (:one)).
;;;
;;; %GET-SUBCLASS-NAMES (class-ref)                                  [FUNCTION]
;;;    takes a class reference and return the list of synonyms for all
;;;    subclasses, ~ including the referenced class.
;;;    Arguments:
;;;    class-ref: a class reference, e.g. "address"
;;;    Return;
;;;    nil or the list of all subclass names.
;;;
;;; %GET-SUBCLASSES (obj-id)                                         [FUNCTION]
;;;    returns the list of subclasses of obj-id.
;;;    Argument:
;;;    obj-id: a class identifier
```

```
;;;    Return:
;;;    nil or the transitive closure along the inverse $IS-A property.
;;;
;;; %GET-TP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref          [FUNCTION]
;;;                                   &key (context *context*))
;;;    Determines the right attribute for the specific class. Throws to
;;;    :error when ~ something is wrong. When class is *none* (orphans) all
;;;    attributes are possible. Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context default current
;;;    Return:
;;;    the attribute attached to the class or nil.
;;;
;;; %GET-TP-VALUE-RESTRICTIONS (prop-id &key (context *context*))    [FUNCTION]
;;;    Obtains the value restrictions from the attribute representation. No
;;;    ~ inheritance is done.
;;;    Restrictions are
;;;       :exists     ; among the possible values one must be of that type
;;;       :forall     ; all values should be of that type
;;;       :type       ; values should be of that type
;;;       :not-type   ; no value should be of that type
;;;       :value      ; only this value is allowed
;;;       :one-of     ; the value should be a member of this list
;;;       :not        ; the value should not be one of this list
;;;       :between    ; the numerical value should be in that range
;;;       :outside    ; the numerical value should be outside that range
;;;    Arguments:
;;;    prop-id: attribute id
;;;    context (opt): context (default current)
;;;    Returns:
;;;    a list of restrictions: e.g. ((:one-of 1 2 3) (:type :integer)
;;;    (:one)).
;;;
;;; %GET-VALUE (obj-id prop-id &key (context *context*))             [FUNCTION]
;;;    Gets the value corresponding to the specified context. To do so uses
;;;    ~ the *version-graph* to locate within a branch the latest valid
;;;    value. ~ The context graph is organized as an a-list of node, each
;;;    node points to ~ its predecessors: ((9 6)(8 3 2)(7 6)...(0)) it
;;;    contains all contexts, ~ context numbers are not necessarily
;;;    increasing along a branch, e.g. if we have ~ fused some contexts.
;;;       Algorithm: (i) gets the branch from the current context, if
;;;    legal, to the root. ~ (the branch is computed only once for a given
;;;    context and cached onto the ~ p-list of the global symbol
;;;    *version-graph*). ~ (ii) goes up in the branch, returning the first
;;;    non nil value associated with ~ a context in the object.
;;;    The exact property of the object must be used, not the generic
;;;    one. %get-value does not return defaults.
;;;       If context is illegal throws to an :error tag.
;;;    Arguments:
```

```
;;;    obj-id: identifier of object
;;;    prop-id: identifier of  property
;;;    context: context
;;;
;;; %%GET-VALUE (obj-id prop-id context)                          [FUNCTION]
;;;    Returns the value associated with a particular context using the
;;;    *version- ~ graph*. If context is illegal throws to an :error tag.
;;;       Like %get-value but does not require the object to have a
;;;    property $TYPE. object-id must not be a REF object, but a regular
;;;    object. Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of local property
;;;    context: context
;;;    Return: the value associated with prop-id.
;;;
;;; %%GET-VALUE-AND-CONTEXT (obj-id prop-id)                      [FUNCTION]
;;;    Done to obtain value for objects which must not be versioned, like
;;;    counters ~ for entities.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    prop-id: local property identifier.
;;;    Retrurn:
;;;    a list whose car is context and cadr is the value.
;;;
;;; %GET-VALUE-FROM-CLASS-REF (class-ref expr)                    [FUNCTION]
;;;    returns the value associated with class-ref from a MOSS pattern.
;;;    Arguments:
;;;    class-ref: e.g. "address"
;;;    Returns:
;;;    value or nil.
;;;
;;; %GET-VALUE-FROM-PROP-REF (prop-ref expr                       [FUNCTION]
;;;                           &key (package *package*))
;;;    returns the value associated with prop-ref from a MOSS pattern.
;;;    Arguments:
;;;    prop-ref: e.g. "town"
;;;    package (key): package in which property has been defined
;;;    Returns:
;;;    value or nil.
;;;
;;; %GET-WORDS-FROM-TEXT (text delimiters)                        [FUNCTION]
;;;    takes an input string and a set of delimiters and returns a list of
;;;    words ~ separated by delimiters.
;;;    Arguments:
;;;    text: a string
;;;    delimiters: a set of characters that delimit a word
;;;    Return:
;;;    a list of words.
;;;
;;; %HAS-ID-LIST (entry inv-id class-id &key (context *context*))    [FUNCTION]
```

```
;;;    Returns the list of all objects ~
;;;         corresponding to entry point entry with inverse terminal
;;;    property inv-id ~ and of class class-id.
;;;       E.g. (%has-id-list 'ATTRIBUTE '$PNAM.OF '$ENT) returns ($EPT) ~
;;;         class-id must be specified locally; i.e., will not return
;;;    objects ~ that are instances of subclasses of class-id.
;;;       If context is illegal throws to an :error tag
;;;    Arguments:
;;;    entry: entry symbol
;;;    inv-id: local inverse attribute
;;;    class-id: identifier of target class
;;;    context: context
;;;    Return:
;;;    a list of objects.
;;;
;;; %%HAS-INVERSE-PROPERTIES (obj-id context)                        [FUNCTION]
;;;    Returns the list of inverse properties local to an object and having
;;;    ~ associated values in the specified context
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-PROPERTIES (obj-id context)                                [FUNCTION]
;;;    Returns the list of properties local to an object and having
;;;    associated ~ values in the specified context, looking directly at the
;;;    internal format ~ of the object. $TYPE is removed.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-PROPERTIES+ (obj-id context)                               [FUNCTION]
;;;    Returns the list of ALL properties LOCAL to an object even with no ~
;;;         value, in the specified context, looking directly at the
;;;    internal format ~ of the object.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-STRUCTURAL-PROPERTIES (obj-id context)                     [FUNCTION]
;;;    Returns the list of structural properties local to an object and
;;;    having  ~ associated values in the specified context. $TYPE and $ID
;;;    are removed. Arguments:
;;;    obj-id: object identifier
;;;    context: context
```

```
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-TERMINAL-PROPERTIES (obj-id context)                       [FUNCTION]
;;;    Returns the list of terminal properties local to an object and having
;;;    ~ associated values in the specified context.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-VALUE (obj-id prop-id context &key no-resolve)             [FUNCTION]
;;;    Returns the value list associated to a particular context if recorded
;;;    locally. No checks are done, except for legal context.
;;;    Arguments:
;;;    obj-id: object or ref to object under consideration (can be dead)
;;;    prop-id: property (may be local or inherited)
;;;    context: context (default current)
;;;    no-resolve (key): if true, skip the call to resolve intended to
;;;    solve multiclass belonging
;;;    Return:
;;;    the local value if any.
;;;
```

8.3.8 Functions for Filtering

```
;;; %BUT-MOSS (item-list)                                             [FUNCTION]
;;;    Removes from a list of symbols any symbol that is not a PDM object or
;;;    that belongs ~ to the MOSS system. Thus we should only get PDM
;;;    application objects. Arguments:
;;;    item-list: a list of symbols
;;;    Return:
;;;    a list of application objects.
;;;
;;; %DETERMINE-PROPERTY-ID (obj-id prop-name                          [FUNCTION]
;;;                          &key (context *context*))
;;;    For the object with id obj-id, attempts to determine which of its ~
;;;         properties corresponds  to prop-name in the specified context.
;;;    If the property is present at the object level, then return it,
;;;    otherwise: - if the object has a prototype, try the prototype
;;;         - if the object has a class, look at the class level for
;;;    properties ~ defined at the class level.
;;;         - if the object belongs to several classes try each one in
;;;    turn. Problems are:
;;;    - tree properties: properties are in a tree (normally each node of
;;;    the ~ tree should have the same property name)
;;;    - lattices of prototypes, where a property could be derived from
;;;    another ~ one thru an IS-A relation (not implemented).
;;;    Arguments:
```

```
;;;    obj-id: identifier of the object
;;;    prop-name: name of property to check: e.g. HAS-NAME (must be
;;;    valid, no check) context: context
;;;    Returns:
;;;    a list of normally a single property-id.
;;;
;;; %DETERMINE-PROPERTY-ID-FOR-CLASS (prop-list class-id            [FUNCTION]
;;;                                   &key (context *context*)
;;;                                   inverse)
;;;    Determines the right prop-id for the specified class among a list of
;;;    properties. No checks on arguments. If property is inherited, then we
;;;    return the closest one. If class is *any* returns the list.
;;;    Arguments:
;;;    prop-list: list of property identifiers, e.g. $T-NAME,
;;;    $S-PERSON-HUSBAND.OF class-id: identifier of the class, e.g.
;;;    $E-PERSON (*any* is allowed) context (opt): context default current
;;;    inverse (key): specify that we look for inverse property
;;;    Returns:
;;;    nil or list of property-id
;;;
;;; %FILTER-ALIVE-OBJECTS (object-list &key (context *context*))    [FUNCTION]
;;;    Takes a list of objects and keeps the ones that are alived in the
;;;    specified ~ context.
;;;    Arguments:
;;;    object-list: list of objects
;;;    context (opt): context (default current)
;;;    Return:
;;;    a (possibly empty) list of objects alive in context.
;;;
;;; %REMOVE-REDUNDANT-PROPERTIES (prop-list                         [FUNCTION]
;;;                               &key (context *context*))
;;;    Removes properties that share the same generic property and keep the
;;;    leftmost ~ one.
;;;    Arguments:
;;;    prop-list: list of property ids
;;;    context (opt): context (default current)
;;;    Return:
;;;    a cleaned list of properties.
;;;
;;; %SELECT-BEST-ENTITIES (entity-score-list)                       [FUNCTION]
;;;    select entities with highest score and return the list.
;;;    Arguments:
;;;    entity-score-list: a list like (($E-PERSON.2 0.5)(..)...)
;;;    Returns:
;;;    the list of entities with the highest score.
;;;
```

8.3.9 Functions for Printing

```
;;; %PEP (obj-id &key (version *context*) (offset 30) (stream t))   [FUNCTION]
```

```
;;;    Prints versions of values from a given context up to the root.
;;;       (%pep obj-id version &rest offset)
;;;       For each property prints values in all previous context up to the
;;;    root ~ does not check for illegal context since it is a printing
;;;    function.
;;;
;;; %PEP-PV (value context-branch &optional (stream t) offset)       [FUNCTION]
;;;    Used by %pep, prints ~
;;;         a set of value associated with a property from the root to
;;;    current context. Offset is currently set to 30.
;;;
;;; %PFORMAT (fstring input &rest prop-list)                         [FUNCTION]
;;;    produces a string for printing from a list produced typically by the
;;;    ~ =make-print-string method.
;;;    Arguments:
;;;    input: a-list, e.g. (("name" "Dupond")("first name" "Jean")...)
;;;    fstring: string control format
;;;    prop-list: a list of properties appearing in the a-list. last
;;;    entry can be the specification of a particular package
;;;    Return:
;;;    a string: e.g. "Dupond, Jean"
;;;
```

8.3.10 Functions Dealing with Multilingual Names

```
;;; %MLN? (expr)                                                     [FUNCTION]
;;;    Checks whether a list is a multilingual name Uses global
;;;    *language-tags* variable. ~ nil is not a valid MLN.
;;;    Argument:
;;;    expr: something like (:en "London" :fr "Londres") or (:name :en
;;;    "London") Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-1? (expr)                                                   [FUNCTION]
;;;    Checks whether a list is a multilingual string. Uses global
;;;    *language-tags* variable. Argument:
;;;    expr: something like (:en "London" :fr "Londres")
;;;    Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-ADD-VALUE (mln value language-tag)                          [FUNCTION]
;;;    Adds a value corresponding to a specific language at the end of the
;;;    list. ~ Does not check if value is already there.
;;;    Arguments:
;;;    mln: multilingual name
;;;    value: coerced to a string
;;;    language-tag: must be legal, i.e. part of *language-tags*
;;;    Return:
;;;    the modified mln.
;;;
```

```
;;; %MLN-EQUAL (mln1 mln2 &key language)                           [FUNCTION]
;;;    Checks whether two multilingual-names are equal. They may also be
;;;    strings. ~ When the language key argument is not there uses global
;;;    *language* variable.~ When *language* is unbound, tries all possible
;;;    languages. The presence of ~ language is only necessary if one of the
;;;    arguments is a string. The function uses %string-norm before
;;;    comparing strings. Argument:
;;;    mln1: a multilingual name or a string
;;;    mln2: id.
;;;    language (key): a specific language tag
;;;    Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-EXTRACT-ALL-SYNONYMS (mln)                                [FUNCTION]
;;;    Extracts all synonyms as a list of strings regardless of the
;;;    language. Arguments:
;;;    mln: multilingual name
;;;    Return
;;;    a list of strings.
;;;
;;; %MLN-EXTRACT-MLN-FROM-REF (ref language                        [FUNCTION]
;;;                            &key no-throw-on-error (default :en)
;;;                            &aux $$$)
;;;    takes a ref input symbol string or mln and builds a specific mln
;;;    restricted to ~ the specified language. If language is :all and ref
;;;    is a string or symbol, then ~ language defaults to default. If ref is
;;;    an MLN, then language defaults to default ~ if there, first found
;;;    language otherwise (canonical behavior). Default beavior ~ is enabled
;;;    only if no-throw-on-error option is set to T. Removes also the
;;;    leading :name marker if present in mln. Arguments:
;;;    ref: a string symbol or mln
;;;    language a language tag or :all
;;;    no-throw (key): if true function does not throw on error but
;;;    returns nil default (key): default language tag (default: :EN)
;;;    Result:
;;;    a valid mln or throws to :error
;;;
;;; %MLN-FILTER-LANGUAGE (mln language-tag &key always)            [FUNCTION]
;;;    Extracts from the MLN the synonym string corresponding to specified
;;;    language. ~ If mln is a string or language is :all, then returns mln
;;;    untouched, unless always ~ is true in which case returns the English
;;;    terms if there, otherwise random terms. If language is not legal,
;;;    throws to :error. Arguments:
;;;    mln: a multilingual name (can also be a simple string)
;;;    language-tag: a legal language tag (part of *language-tags*)
;;;    always (key): if t then returns either the English terms or a
;;;    random one Return:
;;;    a string or nil if none.
;;;
;;; %MLN-GET-CANONICAL-NAME (mln &aux names)                       [FUNCTION]
```

```
;;;    Extracts from a multilingual name the canonical name that will be
;;;    used to build ~ an object ID. By default it is the first name
;;;    corresponding to the value of *language*, ~ or else the first name of
;;;    the English entry, or else the name of the list. ~ An error occurs
;;;    when the argument is not multilingual name. Arguments:
;;;    mln: a multilingual name
;;;    Return:
;;;    2 values: a simple string (presumably for building an ID) and the
;;;    language tag. Error:
;;;    error if not a multilingual name.
;;;
;;; %MLN-GET-FIRST-NAME (name-string)                              [FUNCTION]
;;;    Extracts the first name from the name string. Names are separated by
;;;    semi-columns. E.g. "identity card ; papers" returns "identity card "
;;;    Argument:
;;;    name-string
;;;    Return:
;;;    string with the first name.
;;;
;;; %MLN-MEMBER (input-string tag mln)                             [FUNCTION]
;;;    checks whether the input string is one of the synonyms of the mln in
;;;    the ~ language specified by tag. If tag is :all then we check against
;;;    any synonym ~ in any language.
;;;    Arguments:
;;;    input-string: input string (to be normed with %norm-string)
;;;    tag; a legal language tag or :all
;;;    mln: a mltilingual name
;;;    Return:
;;;    non nil value if true.
;;;
;;; %MLN-MERGE (&rest mln)                                         [FUNCTION]
;;;    Takes a list of MLN and produces a single MLN as a result, merging
;;;    the ~ synonyms (using or norm-string comparison) and keeping the
;;;    order of the ~ inputs.
;;;    Arguments:
;;;    mln: a multilingual name
;;;    more-mln (rest): more multilingual names
;;;    Return:
;;;    a single multilingual name
;;;    Error:
;;;    in case one of the mln has a bad format.
;;;
;;; %MLN-NORM (expr)                                               [FUNCTION]
;;;    Norms an mln expr by removing the :name keyword if there.
;;;    Argument:
;;;    expr: expression to test
;;;    Return:
;;;    normed expr, nil is an error)
;;;
;;; %MLN-PRINT-STRING (mln &optional (language-tag :all))          [FUNCTION]
```

```
;;;    Prints a multilingual name nicely.
;;;    Arguments:
;;;    mln: multilingual name
;;;    language-tag: legal language-tag (default :all)
;;;    Result:
;;;    a string ready to be printed (can be empty).
;;;
;;; %MLN-REMOVE-LANGUAGE (mln language-tag)                        [FUNCTION]
;;;    Removes the entry corresponding to a language from a multilingual
;;;    name. Arguments:
;;;    mln: multilanguage name
;;;    language-tag: language to remove (must be legal
;;;    Return:
;;;    modified mln.
;;;
;;; %MLN-REMOVE-VALUE (mln value language-tag)                     [FUNCTION]
;;;    Removes a value in the list of synonyms of a particular language.
;;;    Arguments:
;;;    mln: multilingual name
;;;    value: coerced to a string (loose package context)
;;;    language-tag: must be legal
;;;    Return:
;;;    the modified mln.
;;;
;;; %MLN-SET-VALUE (mln language-tag syn-string)                   [FUNCTION]
;;;    Sets the language part to the specified value, erasing the old value.
;;;    Arguments:
;;;    mln: a multilingual name
;;;    language-tag: language tag
;;;    syn-string: coerced to a string (loose package context)
;;;    Return:
;;;    the modified normed MLN.
;;;
;;; %MULTILINGUAL-NAME? (expr)                                     [FUNCTION]
;;;    Checks whether a list is a multilingual string. Uses global
;;;    *language-tags* variable. Argument:
;;;    expr: something like (:en "London" :fr "Londres")
;;;    Result:
;;;    T if OK, nil otherwise.
;;;
```

8.3.11 Functions Dealing with Strings and Synonyms

```
;;; STR-EQUAL (aa bb)                                              [FUNCTION]
;;;
;;; %STRING-NORM (input &optional (interchar #\-))                 [FUNCTION]
;;;    Same as make-value-string but with more explicit name.
;;;    If input is a multilingual string, uses the canonical name.
;;;    If input is NIL throws to :error
;;;    Arguments:
```

```
;;;    input: may be a string, a symbol or a multilingual string
;;;    interchar: character to hyphenate the final string (defaul is #-
;;;    Return:
;;;    a normed string, e.g. "AU-JOUR-D-AUJOURD-HUI ".
;;;
;;; %SYNONYM-ADD (syn-string value)                          [FUNCTION]
;;;    Adds a value to a string at the end.
;;;    Sends a warning if language tag does not exist and does not add
;;;    value. Arguments:
;;;    syn-string: a synonym string or nil
;;;    value: value to be included (coerced to string)
;;;    Return:
;;;    modified string when no error, original string otherwise.
;;;
;;; %SYNONYM-EXPLODE (text)                                  [FUNCTION]
;;;    Takes a string, consider it as a synonym string and extract items
;;;    separated ~ by a semi-column. Returns the list of string items.
;;;    Arguments:
;;;    text: a string
;;;    Return
;;;    a list of strings.
;;;
;;; %SYNONYM-MAKE (&rest item-list)                          [FUNCTION]
;;;    Builds a synonym string with a list of items . Eadh item is coerced
;;;    to a string. Arguments:
;;;    item-list: a list of items
;;;    Return:
;;;    a synonym string.
;;;
;;; %SYNONYM-MEMBER (value text)                             [FUNCTION]
;;;    Checks if a string is part of the synonym list. Uses the %string-norm
;;;    ~ function to normalize the strings before comparing.
;;;    Arguments:
;;;    value: a value coerced to a string
;;;    text: a synonym string
;;;    Return:
;;;    a list of unnormed remaining synonyms if succcess, NIL if nothing
;;;    left.
;;;
;;; %SYNONYM-MERGE-STRINGS (&rest names)                     [FUNCTION]
;;;    Merges several synonym string, removing duplicates (using a
;;;    norm-string ~ comparison) and preserving the order.
;;;    Arguments:
;;;    name: a possible complex string
;;;    more-names (rest): more of that
;;;    Return:
;;;    a single complex string
;;;    Error:
;;;    when some of the aguments are not strings.
;;;
```

```
;;; %SYNONYM-REMOVE (value text)                                  [FUNCTION]
;;;    Removes a synonym from the list of synonyms. If nothing is left
;;;    return ~ empty string.
;;;    Arguments:
;;;    value: a value coerced to a string
;;;    text: a synonym string
;;;    Return:
;;;    the string with the value removed NIL if nothing left.
;;;
```

8.3.12 Miscellaneous

```
;;; %CLEAN-WORD-LIST (word-list &rest empty-word-lists)           [FUNCTION]
;;;    remove empty words from the list using empty-word lists.
;;;    Arguments:
;;;    word-list: list of words to be cleaned
;;;    empty-word-lists (rest): lists of empty words
;;;    Returns:
;;;    the cleaned list.
;;;
;;; %INTERSECT-SYMBOL-LISTS (ll1 ll2)                             [FUNCTION]
;;;    Intersect two lists of symbols by using their p-list. Done in linear
;;;    time.
;;;
;;; %RANK-ENTITIES-WRT-WORD-LIST (entity-list word-list)          [FUNCTION]
;;;    takes a list of entity ids and a list of words, and returns a score
;;;    for ~ each entity that is function of the number of apparitions of
;;;    the words in ~ attributes of the entity.
;;;    Arguments:
;;;    entity-list: a list of object ids (ei)
;;;    word-list: a list of words (wordj)
;;;    Returns:
;;;    a list of pairs (ei wi), e.g. (($e-person.2 0.75)($E-PERSON.3
;;;    0.5)).
;;;
```

8.4 Service Functions by Alphabetical Order

```
;;;
;;; %ACCESS-ENTITIES-FROM-WORD-LIST (word-list                    [FUNCTION]
;;;                                   &rest empty-word-lists)
;;;    takes a list of words, removes the empty words and tries to find
;;;    entities ~ of the KB using entry points from the list of words and
;;;    filtering the results ~ to keep only best matches.
;;;    Arguments:
;;;    word-list: input list of words
;;;    empty-word-list (key): list of empty words to remove from word
;;;    list Return:
;;;    list of object ids or nil.
;;;
```

```
;;; %%ADD-VALUE (obj-id prop-id value context)                    [FUNCTION]
;;;    Inserts a new value for prop-id, no checking done on arguments
;;;    Context must be ~ explicitly given.
;;;    Arguments:
;;;    obj-id: id of object
;;;    prop-id: id of property for which we add the value
;;;    value: value to add
;;;    context: context in which we add the value
;;;    Return:
;;;    the list representing the modified object.
;;;
;;; %ADD-VALUE (obj-id prop-id value &key (context *context*)        [FUNCTION]
;;;             before-value allow-duplicates)
;;;    Inserts a new value for a given property in the specified context. ~
;;;         Copies the old values from a previous context if necessary.
;;;    We assume that prop-id is the correct local property, and that ~
;;;         the value has been validated.
;;;    Arguments:
;;;    obj-id: id of the object to be modified
;;;    prop-id: id of the target property
;;;    value: value to be added
;;;    context: context in which the value is added
;;;    before-value (key): value - specifies that we want to insert in
;;;    front of value allow-duplicates (key): t - allows to insert duplicate
;;;    values. No cardinality checking done on args. If value is there does
;;;    not do ~ anything.
;;;    Throws to an :error tag if the context is not allowed
;;;    Return:
;;;    the internal format of the modified object
;;;
;;; %%ALIVE? (obj-id context)                                       [FUNCTION]
;;;    Checks if object is PDM, *context* is OK, and object is alive. If
;;;    not, throws ~ to error.
;;;    Arguments:
;;;    obj-id: id
;;;    context (opt): context (default current)
;;;    Return:
;;;    resoved id, if OK, throws to :error otherwise.
;;;
;;; %ALIVE? (obj-id &key (context *context*))                       [FUNCTION]
;;;    Checks if object exists in given context ~
;;;         each object has a tombstone which records if it has been killed
;;;    ~ by somebody in the given context. This is a source of inconsistency
;;;    ~ since if the tombstone exists then the properties must be removed
;;;    in ~ current context. I.e. one cannot have a link onto an object
;;;    which ~ has been killed.
;;;    If context is illegal throws to :error.
;;;    Arguments:
;;;    obj-id: identifier of the object to be checked
;;;    context (opt): context (default current context)
```

```
;;;    Return:
;;;    nil if object is dead, resolved id otherwise
;;;
;;; %%ALLOWED-CONTEXT? (context)                                [FUNCTION]
;;;    Checks if the context is allowed. Does this by looking at the global
;;;    variable *version-graph*. If not allowed throws to :error.
;;;
;;; %ANCESTOR? (obj1 obj2 &key (prop-id '$is-a)                 [FUNCTION]
;;;              (context *context*))
;;;    Checks if obj2 is equal to obj1, or belongs to the transitive closure
;;;    of obj1 ~ using property prop-id.
;;;    Arguments:
;;;    obj1: first object
;;;    obj2: second object
;;;    prop-id: property to follow (default is $IS-A)
;;;    context (opt): context (default current)
;;;
;;; %BUILD-METHOD (function-name arg-list body)                 [FUNCTION]
;;;    build a MOSS method from its definition in the various defxxxmethof
;;;    macros. ~ Body may have a comment (string) and/or a declare
;;;    statement. The method ~ body is enclosed in a catch :return clause to
;;;    allow exits, since we cannot ~ use the return-from clause inside the
;;;    method. When the compiler is included in the environment, compiles
;;;    the code. Arguments:
;;;    function-name: the name of the function implementing the method
;;;    (e.g. $UNI=I=0=SUMMARY) arg-list: method arguments
;;;    body: the body of the function
;;;    Return:
;;;    the function name.
;;;
;;; %BUT-MOSS (item-list)                                       [FUNCTION]
;;;    Removes from a list of symbols any symbol that is not a PDM object or
;;;    that belongs ~ to the MOSS system. Thus we should only get PDM
;;;    application objects. Arguments:
;;;    item-list: a list of symbols
;;;    Return:
;;;    a list of application objects.
;;;
;;; %CLASSLESS-OBJECT? (object-id &key (context *context*))     [FUNCTION]
;;;    Checks if the entity is a classless object, i.e. its type is *none*
;;;    Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    Return:
;;;    T if OK, NIL otherwise.
;;;
;;; %CLEAN-WORD-LIST (word-list &rest empty-word-lists)         [FUNCTION]
;;;    remove empty words from the list using empty-word lists.
;;;    Arguments:
;;;    word-list: list of words to be cleaned
```

```
;;;    empty-word-lists (rest): lists of empty words
;;;    Returns:
;;;    the cleaned list.
;;;
;;; %CREATE-BASIC-NEW-OBJECT (object-id &key (context *context*) id  [FUNCTION]
;;;                           ideal (package *package*))
;;;    Creates a skeleton of object containing only the $TYPE and $ID ~
;;;         properties in the current context.
;;;    Arguments:
;;;    object-id: presumably class of the object to be created
;;;    context (key): context default current
;;;    id (key): if there, id is specified (we do not create it)
;;;    ideal (key): if t, we want to create an ideal instance
;;;                        (with sequence number 0)
;;;    package (key): package (default is current)
;;;    Return:
;;;    id of the newly created object.
;;;
;;; %CREATE-BASIC-ORPHAN (&key (context *context*)                   [FUNCTION]
;;;                       (package *package*))
;;;    Creates a skeleton for an orphan object and the associated key.
;;;    Arguments:
;;;    context (key): context (default current)
;;;    Return:
;;;    the id of the new orphan.
;;;
;;; %CREATE-ENTRY-POINT-METHOD (object-id object-name                [FUNCTION]
;;;                             &optional args doc body)
;;;    Creates an entry point method, with default if no arguments. ~
;;;         the =make-entry symbol is defined in the same package as
;;;    object-id. If args is present it must contain an option &key (package
;;;    *package*). Arguments:
;;;    object-id: id to which the =make-entry method will be attached
;;;    object-name: name of the object (for error message)
;;;    args (opt): arguments of the method (default is (value-list))
;;;    doc (opt): documentation string
;;;    body (opt): body of the method
;;;    Return:
;;;    an own-method id.
;;;
;;; %DETERMINE-PROPERTY-ID (obj-id prop-name                         [FUNCTION]
;;;                         &key (context *context*))
;;;    For the object with id obj-id, attempts to determine which of its ~
;;;         properties corresponds  to prop-name in the specified context.
;;;    If the property is present at the object level, then return it,
;;;    otherwise: - if the object has a prototype, try the prototype
;;;         - if the object has a class, look at the class level for
;;;    properties ~ defined at the class level.
;;;         - if the object belongs to several classes try each one in
;;;    turn. Problems are:
```

```
;;;    - tree properties: properties are in a tree (normally each node of
;;;    the ~ tree should have the same property name)
;;;    - lattices of prototypes, where a property could be derived from
;;;    another ~ one thru an IS-A relation (not implemented).
;;;    Arguments:
;;;    obj-id: identifier of the object
;;;    prop-name: name of property to check: e.g. HAS-NAME (must be
;;;    valid, no check) context: context
;;;    Returns:
;;;    a list of normally a single property-id.
;;;
;;; %DETERMINE-PROPERTY-ID-FOR-CLASS (prop-list class-id             [FUNCTION]
;;;                                   &key (context *context*)
;;;                                   inverse)
;;;    Determines the right prop-id for the specified class among a list of
;;;    properties. No checks on arguments. If property is inherited, then we
;;;    return the closest one. If class is *any* returns the list.
;;;    Arguments:
;;;    prop-list: list of property identifiers, e.g. $T-NAME,
;;;    $S-PERSON-HUSBAND.OF class-id: identifier of the class, e.g.
;;;    $E-PERSON (*any* is allowed) context (opt): context default current
;;;    inverse (key): specify that we look for inverse property
;;;    Returns:
;;;    nil or list of property-id
;;;
;;; %ENTRY? (object-id &key (context *context*)                     [FUNCTION]
;;;         (package *package* package-here?))
;;;    Checks if the id is that of an entry point.
;;;    Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    package (opt): if obj-id is a symbol, then that of obj-id default
;;;    current Returns:
;;;    nil or something not meaningful
;;;
;;; %EXPLODE-OBJECT (obj-id &key into)                              [FUNCTION]
;;;    function that explodes an instance into a set of triples appending it
;;;    to the ~ list specified by the into variable.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    into (key): list to which to append new triples
;;;    Return:
;;;    the list of triples.
;;;
;;; %EXTRACT (entry tp class &key (context *context*)               [FUNCTION]
;;;          (filter nil filter-there?)
;;;          (class-ref nil class-there?)
;;;          (package *package* package-there?)
;;;          allow-multiple-values)
;;;    Extracts an object that is specified by its entry-point - attribute -
```

```
;;;    ~ concept - and context. When more than one, applies a function when
;;;    ~ specified in the option <:filter function>. If more than one value
;;;    left ~ then an error is declared, unless they are explicitely allowed
;;;    by setting the ~ keyword argument :allow-multiple-values to T.
;;;    Arguments:
;;;    entry: entry point (string, symbol or mln)
;;;    tp: attribute (string, symbol or mln)
;;;    class: class name (string, symbol or mln)
;;;    context (key): context default current
;;;    filter (key): filter function when more than one value is expected
;;;    filter-there?: variable set to T when a filter is specified
;;;    class-ref (key): reference of class to locate a property
;;;    class-there?: variable set to T when class-ref is specified
;;;    package (key): default current
;;;    allow-multiple-values (key): if T the function may return more
;;;    than one value Returns
;;;    nil if it cannot find anything
;;;    a symbol  for a single object
;;;    a list when it finds several objects and multiples are allowed
;;;
;;; %EXTRACT-FROM-ID (entry prop-id class-id                         [FUNCTION]
;;;                  &key (context *context*))
;;;    Returns the list of all objects  corresponding to entry point entry ~
;;;         with inverse attribute of prop-id and of class class-id.
;;;       E.g. (%extract-from-id 'ATTRIBUTE '$ENAM '$ENT 0) returns ($EPT)
;;;       obj-id are returned if they are in class class-id or in one of
;;;    the subclasses ~ (transitive closure along the $IS-A.OF link).
;;;       Also all meta-models $ENT, $EPS, $EPT, $FN, $CTR ($EIL?) are
;;;    defined ~ as instances of $ENT.
;;;       If context is illegal throws to an :error tag.
;;;       The function works with the *any* and *none* pseudo classes:
;;;     (%extract-from-id <entry> <prop-id> '*any*) returns everything
;;;     (%extract-from-id <entry> <prop-id> '*none*) returns only classless
;;;    objects. We also allow objects with multiple classes.
;;;    No checks are done to speed execution.
;;;    Arguments:
;;;    entry: entry point (symbol)
;;;    prop-id: id of property to be used to make the inverse
;;;                  should be the exact property associated with class-id
;;;    class-id: id of the potential class of the object
;;;    context (key): context (default current)
;;;    Returns:
;;;    a list of objects maybe one
;;;
;;; %FILTER-ALIVE-OBJECTS (object-list &key (context *context*))     [FUNCTION]
;;;    Takes a list of objects and keeps the ones that are alived in the
;;;    specified ~ context.
;;;    Arguments:
;;;    object-list: list of objects
;;;    context (opt): context (default current)
```

```
;;;    Return:
;;;    a (possibly empty) list of objects alive in context.
;;;
;;; %GET-ALL-ATTRIBUTES "()"                                    [FUNCTION]
;;;    Gets the list of all attributes in all contexts. No arguments.
;;;
;;; %%GET-ALL-CLASS-ATTRIBUTES (class-id &key (context *context*))   [FUNCTION]
;;;    Collects all attributes starting with class-id through superclasses.
;;;    Attributes of the same property tree can be included in the
;;;    result. Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-INVERSE-RELATIONS (class-id                 [FUNCTION]
;;;                                   &key (context *context*))
;;;    Collects all relations starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-PROPERTIES (class-id &key (context *context*))   [FUNCTION]
;;;    Collects all properties starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-CLASS-RELATIONS (class-id &key (context *context*))    [FUNCTION]
;;;    Collects all relations starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %GET-ALL-CONCEPTS "()"                                      [FUNCTION]
;;;    Gets the list of all concepts in all contexts. No arguments.
;;;
;;; %GET-ALL-ENTRY-POINTS "()"                                  [FUNCTION]
;;;    Gets the list of all entry-points in all contexts. No arguments.
;;;
;;; %GET-ALL-INSTANCES (class-id &key package)                  [FUNCTION]
```

```
;;;    Gets the list of all instances of a class in all contexts. We assume
;;;    that ~ counter is in context 0.
;;;    The default behavior is that the package for the instances is the
;;;    same as ~ the package of the class-id symbol. Otherwise one should
;;;    know what ~ one does...
;;;    Arguments:
;;;    class-id: id of the class
;;;    return:
;;;    list of all bound identifiers of the instances of the class
;;;
;;; %GET-ALL-INVERSE-RELATIONS "()"                              [FUNCTION]
;;;    Gets the list of all inverse relations in all contexts. No arguments.
;;;
;;; %GET-ALL-METHODS "()"                                        [FUNCTION]
;;;    Gets the list of all methods in all contexts. No arguments.
;;;
;;; %GET-ALL-MOSS-OBJECTS "()"                                   [FUNCTION]
;;;    Gets the list of all MOSS objects in all contexts. No arguments.
;;;
;;; %GET-ALL-ORPHANS "()"                                        [FUNCTION]
;;;    Gets the list of all orphans in all contexts. No arguments.
;;;
;;; %GET-ALL-RELATIONS "()"                                      [FUNCTION]
;;;    Gets the list of all relations in all contexts. No arguments.
;;;
;;; %%GET-ALL-SUBCLASS-INVERSE-RELATIONS (class-id               [FUNCTION]
;;;                                   &key (context *context*))
;;;    Collects all properties starting with class-id through superclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %%GET-ALL-SUBCLASS-RELATIONS (class-id                       [FUNCTION]
;;;                               &key (context *context*))
;;;    Collects all relations starting with class-id through subclasses.
;;;    Relations of the same property tree can be included in the result.
;;;    Arguments:
;;;    class-id: identifier of the class
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of class attributes.
;;;
;;; %GET-ALL-SYMBOLS "()"                                        [FUNCTION]
;;;    Gets the list of all symbols in all contexts (counters, global
;;;    variables,... No arguments.
;;;
;;; %GET-ALL-VERSIONS (obj-id prop-id)                           [FUNCTION]
```

```
;;;    Returns all versions of values or of successors associated with ~
;;;         the given property prop-id. Removes ~
;;;         duplicates of the values corresponding to the various versions.
;;;    ~ The function is used when trying to get all objects of a kind ~
;;;         from the system, e.g. by %get-all-symbols.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    prop-id: property id for which we want all values
;;;    Return:
;;;    a list of all values regardless of versions.
;;;
;;; %GET-AND-INCREMENT-COUNTER (obj-id                              [FUNCTION]
;;;                             &optional (counter-link-id '$ctrs))
;;;    Obtains the value of a counter associated with obj-id and increases
;;;    this value ~ in the counter itself. Throws to :error if obj-id is
;;;    invalid. Arguments
;;;    obj-id: id of the object that owns the counter (ususally a class)
;;;    counter-link-id (opt): name of the property linking to the counter
;;;    (default $CTRS) Return:
;;;    old value of the counter.
;;;
;;; %GET-APPLICATION-CLASSES (&key (context *context*))             [FUNCTION]
;;;    returns the list of classes of a given application by extracting them
;;;    from ~ the system object and removing the system classes.
;;;    If OMAS is active, then we consider objects from the current
;;;    package. Arguments:
;;;    context (opt): default is *context*
;;;    Returns:
;;;    a list of class ids.
;;;
;;; %GETC (obj-id prop-id context)                                  [FUNCTION]
;;;    Retrieves the value associated with prop-id ~
;;;         for the specified context from the p-list of obj-id used as a
;;;    cache. ~ If the context is illegal, then throws to an :error tag.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of local property
;;;    context: context
;;;    Returns:
;;;    nil or the cached value, Throws to :error when context is illegal
;;;    or object dead.
;;;
;;; %GET-CLASS-CANONICAL-NAME-FROM-REF (ref)                        [FUNCTION]
;;;    uses ref as an entry point to the property. If it fails return nil.
;;;    Otherwise, ~ returns the canonical name in the current language.
;;;    Arguments:
;;;    ref: e.g. "country"
;;;    Return:
;;;    canonical name in the current language specified by *language*.
;;;
```

```
;;; %GET-CLASS-ID-FROM-REF (class-ref &key (context *context*)       [FUNCTION]
;;;                         (package *package*))
;;;    Retrieves the class id from its name. Should be unique. Error
;;;    otherwise. Argument:
;;;    class-ref: a symbol, string or multilingual name, e.g. PERSON,
;;;    "Person", (:en ...) context (opt): context default current
;;;    package (opt): default current
;;;    Return:
;;;    class-id, e.g. $E-PERSON
;;;
;;; GET-CURRENT-YEAR "()"                                            [FUNCTION]
;;;
;;; %GET-DEFAULT-FROM-CLASS (prop-id class-id)                       [FUNCTION]
;;;    gets the default value-list associated with a property in a given
;;;    class. ~ The way to do it is to ask the property itself; then the
;;;    ideal of the class. Defaults cannot be inherited.
;;;    Arguments:
;;;    prop-id: id of corresponding property
;;;    class-id: id of the class
;;;    Return:
;;;    defaul value or nil.
;;;
;;; %GET-OBJECT-CLASS-NAME (obj-id)                                  [FUNCTION]
;;;    get the class names of an object.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    Return:
;;;    a list of strings representing the names of the classes to which
;;;    the object belongs.
;;;
;;; %GET-EMPTY-WORDS (language)                                      [FUNCTION]
;;;    recover a list of empty words in specified language. This assumes ~
;;;             that a class empty-words exists and has been initialized.
;;;    Arguments:
;;;    language: a string or keyword
;;;    Return:
;;;    a list of empty words or nil.
;;;
;;; %GET-GENERIC-PROPERTY (prop-list &key (context *context*))       [FUNCTION]
;;;    We start with a set of properties having the same name and try to
;;;    obtain the ~ one that is generic, i.e. that does not have an ancestor
;;;    (no $IS-A property). Argument:
;;;    prop-list: list of property identifiers
;;;    Return:
;;;    the identifier of the generic property or nil. Warns if there are
;;;    more than one left.
;;;
;;; %GET-GENERIC-PROPERTY-FROM-NAME (prop-name                       [FUNCTION]
;;;                                   &key (context *context*))
;;;    We start with a property name and try to obtain the id of the ~
```

```
;;;         one that is generic, i.e. that does not have an ancestor.
;;;    Argument:
;;;    prop-name: property name
;;;    Return:
;;;    the identifier of the generic property or nil. Warns if there are
;;;    more than one left.
;;;
;;; %GET-GENERIC-PROPERTY-FROM-REF (prop-ref)                        [FUNCTION]
;;;    We start with a set of properties having the same name and try to
;;;    obtain the ~ one that is generic, i.e. that does not have an ancestor
;;;    (no $IS-A property). Argument:
;;;    prop-ref: string or symbol e.g. "content" or CONTENT
;;;    Return:
;;;    the identifier of the generic property (e.g. HAS-CONTENT) or nil.
;;;
;;; %GET-INDEX-WEIGHTS (task-id)                                     [FUNCTION]
;;;    get a list of pairs index (a string) value from the list of ~
;;;        index patterns associated with the task.
;;;    Arguments:
;;;    task: a task object
;;;    Return:
;;;    a list of pairs.
;;;
;;; %GET-INSTANCE-COUNT (class-id &key (context *context*)           [FUNCTION]
;;;                       (allow-subclasses t))
;;;    Gets an upper-bound on the number of instances for a given class.
;;;    Arguments:
;;;    class-id: identifier of class
;;;    allow-subclasses (key): if T counts instances of subclasses
;;;    Returns:
;;;    a number
;;;
;;; %%GET-LIST-OF-VALUES-AND-CONTEXT (obj-id prop-id)                [FUNCTION]
;;;    Done to obtain raw value list of objects.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: property local identifier
;;;    Returns:
;;;    a list of all values by context (internal format of the set of
;;;    values).
;;;
;;; %GETM (object-id method-name context &rest own-or-instance)      [FUNCTION]
;;;    When methods are compiled on p-list it is done as follows:
;;;     (=print-self (:own (nn <int-function name>)*) (:instance  <...>))
;;;       if the method type is not specified then the format is slightly ~
;;;         different from the more general format.
;;;       The version must be recorded locally otherwise we shall try ~
;;;         to inherit it. Remember this is a cache, not an object.
;;;       When using versions we can have the method cached onto the p-list
;;;    for a ~ previous version. If the method has not been modified since,
```

```
;;;    it is a waste ~ of energy to recover it again, to recompile it, and
;;;    to give it a new name. ~ Rather, we want to check that it was not
;;;    modified along the same branch indeed, ~ then we cache it under the
;;;    current version explicitely with the old internal ~ name. In order to
;;;    do that, we must record every time a method code or arguments ~ was
;;;    changed. Doing
;;;           (%get-value <method-id> <modif> *current-context*)
;;;       will return the last modification or nil (it could have been
;;;    modified, but ~ then it is not in the same branch.
;;;       So for recovering a cached method, we first get the current
;;;    branch of the ~ path from the current context to the root of the
;;;    version graph. We then loop on the successive values of the versions
;;;    in the branch starting ~ with the current one. At each step, either
;;;    we get a function, or this is the ~ step at which the version was
;;;    modified, in which case we loose and we must ~ recompute and
;;;    recompile the method. Otherwise, when we get a previously ~ recorded
;;;    method, we must cache it explicitely on the p-list for next time
;;;    around. Arguments: object-id: id of the object
;;;    method-name: name of the target method
;;;    context: context
;;;    own-or-instance (key): flag to specify type of method
;;;    Return:
;;;    method or nil
;;;
;;; %GET-NAME-FROM-REF (ref object-type)                           [FUNCTION]
;;;    Takes a reference and tries to obtain the official name for the
;;;    object, e.g. ~ "class" -> CONCEPT
;;;    If ref is a symbol, we assume it is the name we are looking for,
;;;       if ref is a multilingual name, we use the canonical name,
;;;       if ref is a string, we build a name from it, specific to the
;;;    object type, ~ and recover the canonical name from its recorded name.
;;;    if the object is a property yet undefined, we return the cooked up
;;;    name. Arguments:
;;;    ref: a symbol, string or multilingual name
;;;    object-type: a keyward giving the type of the object (:class,
;;;    :attribute, ...) package (opt): a package spec for the name
;;;    Return:
;;;    the canonical name of the object.
;;;
;;; %%GET-OBJECTS-FROM-ENTRY-POINT (entry &key (context *context*)   [FUNCTION]
;;;                                 keep-entry)
;;;    Sets up an a-list (inv-prop obj-id) of all objects for which entry is
;;;    an ~ entry point, eliminating the current-system (inv prop:$EPLS.OF).
;;;    Arguments:
;;;    entry: entry-point
;;;    context (key): context default current
;;;    keep-entry (key): if t keep the entry point
;;;    Return:
;;;    a list of pairs (<inv-prop-id> . <obj-id>)  or nil
;;;
```

```
;;; %GET-PROP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref        [FUNCTION]
;;;                                       &key (context *context*)
;;;                                       (package *package*))
;;;    Determines the right property for the specific class. Throws to
;;;    :error when ~ something is wrong. Property may be inherited.
;;;    Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context default current
;;;    package (opt): defualt current
;;;    Return:
;;;    the property attached to the class.
;;;
;;; %GET-PROPERTIES (object-id &key (context *context*))             [FUNCTION]
;;;    Obtain all the possible properties for a given object. If an orphan,
;;;    then ~ we only get local properties attached to the orphan.
;;;    Otherwise, we get ~ all inherited properties corresponding to the
;;;    object class. Arguments:
;;;    object-id: identifier of object
;;;    context (opt): context (default current)
;;;    Return:
;;;    a list of properties associated with the object.
;;;
;;; %GET-PROPERTY-CANONICAL-NAME-FROM-REF (ref)                      [FUNCTION]
;;;    uses ref as an entry point to the property. If it fails return nil.
;;;    Otherwise, ~ returns the canonical name in the current language.
;;;    Arguments:
;;;    ref: e.g. "country"
;;;    Return:
;;;    canonical name in the current language specified by *language*.
;;;
;;; %GET-PROPERTY-ID-FROM-NAME (obj-id prop-ref                      [FUNCTION]
;;;                             &key (context *context*)
;;;                             (package *package*))
;;;    Function that has the difficult role of finding the right property
;;;    that ~ applies to the object, and corresponds to the prop-name. This
;;;    is ~ difficult in case we have tree-properties.
;;;    It is used by elementary universal methods like =get =put =delete
;;;    Result is cached onto the p-list of the property name (e.g.
;;;    HAS-NAME). Arguments:
;;;    obj-id: object for which we try to determine property
;;;    prop-ref: property ref, e.g. HAS-FIRST-NAME, "first name", or mln
;;;    context (opt): context default current
;;;    Return:
;;;    the identifier of the corresponding property or nil if the class
;;;    inherits property.
;;;
;;; %GET-PROPERTY-SYNONYMS (prop-ref &key (package *package*))       [FUNCTION]
;;;    takes a property reference and return the list of synonyms for this
;;;    property. Arguments:
```

```
;;;    prop-ref: a property reference, e.g. "street"
;;;    package (key): package in which property name has been defined
;;;    Return;
;;;    nil or the list of all property synonyms.
;;;
;;; %GET-RELEVANT-WEIGHTS (weight-list word-patterns                 [FUNCTION]
;;;                         &key used-patterns)
;;;    function to obtain weigths for a list of words.
;;;    Arguments:
;;;    weight-list: list of weights e.g. ((<string> .6))
;;;    word-patterns: the list of expressions (strings) to test
;;;    used-patterns (key): expressions that have already been tried
;;;    Return:
;;;    a list of pairs string weight.
;;;
;;; %GET-SP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref          [FUNCTION]
;;;                                      &key (context *context*)
;;;                                      (package *package*))
;;;    Determines the right relation for the specific class. Throws to
;;;    :error when ~ something is wrong.
;;;    Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    the property attached to the class.
;;;
;;; %GET-SP-VALUE-RESTRICTIONS (prop-id &key (context *context*))    [FUNCTION]
;;;    Obtains the value restrictions from the attribute representation. No
;;;    ~ inheritance is done.
;;;    Restrictions are
;;;       :exists     ; among the possible values one must be of that type
;;;       :forall     ; all values should be of that type
;;;       :type       ; values should be of that type
;;;       :not-type   ; no value should be of that type
;;;       :value      ; only this value is allowed
;;;       :one-of     ; the value should be a member of this list
;;;       :not        ; the value should not be one of this list
;;;    Arguments:
;;;    prop-id: attribute id
;;;    context (opt): context (default current)
;;;    Returns:
;;;    a list of restrictions: e.g. ((:one-of 1 2 3) (:type :integer)
;;;    (:one)).
;;;
;;; %GET-SUBCLASS-NAMES (class-ref)                                  [FUNCTION]
;;;    takes a class reference and return the list of synonyms for all
;;;    subclasses, ~ including the referenced class.
;;;    Arguments:
;;;    class-ref: a class reference, e.g. "address"
```

```
;;;    Return;
;;;    nil or the list of all subclass names.
;;;
;;; %GET-SUBCLASSES (obj-id)                                       [FUNCTION]
;;;    returns the list of subclasses of obj-id.
;;;    Argument:
;;;    obj-id: a class identifier
;;;    Return:
;;;    nil or the transitive closure along the inverse $IS-A property.
;;;
;;; %GET-TP-ID-FROM-NAME-AND-CLASS-REF (prop-name class-ref        [FUNCTION]
;;;                                     &key (context *context*))
;;;    Determines the right attribute for the specific class. Throws to
;;;    :error when ~ something is wrong. When class is *none* (orphans) all
;;;    attributes are possible. Arguments:
;;;    prop-name: name of a property
;;;    class-ref: class name or class identifier
;;;    context (opt): context default current
;;;    Return:
;;;    the attribute attached to the class or nil.
;;;
;;; %GET-TP-VALUE-RESTRICTIONS (prop-id &key (context *context*))    [FUNCTION]
;;;    Obtains the value restrictions from the attribute representation. No
;;;    ~ inheritance is done.
;;;    Restrictions are
;;;       :exists     ; among the possible values one must be of that type
;;;       :forall     ; all values should be of that type
;;;       :type       ; values should be of that type
;;;       :not-type   ; no value should be of that type
;;;       :value      ; only this value is allowed
;;;       :one-of     ; the value should be a member of this list
;;;       :not        ; the value should not be one of this list
;;;       :between    ; the numerical value should be in that range
;;;       :outside    ; the numerical value should be outside that range
;;;    Arguments:
;;;    prop-id: attribute id
;;;    context (opt): context (default current)
;;;    Returns:
;;;    a list of restrictions: e.g. ((:one-of 1 2 3) (:type :integer)
;;;    (:one)).
;;;
;;; %GET-VALUE (obj-id prop-id &key (context *context*))           [FUNCTION]
;;;    Gets the value corresponding to the specified context. To do so uses
;;;    ~ the *version-graph* to locate within a branch the latest valid
;;;    value. ~ The context graph is organized as an a-list of node, each
;;;    node points to ~ its predecessors: ((9 6)(8 3 2)(7 6)...(0)) it
;;;    contains all contexts, ~ context numbers are not necessarily
;;;    increasing along a branch, e.g. if we have ~ fused some contexts.
;;;       Algorithm: (i) gets the branch from the current context, if
;;;    legal, to the root. ~ (the branch is computed only once for a given
```

```
;;;    context and cached onto the ~ p-list of the global symbol
;;;    *version-graph*). ~ (ii) goes up in the branch, returning the first
;;;    non nil value associated with ~ a context in the object.
;;;    The exact property of the object must be used, not the generic
;;;    one. %get-value does not return defaults.
;;;       If context is illegal throws to an :error tag.
;;;    Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of  property
;;;    context: context
;;;
;;; %%GET-VALUE (obj-id prop-id context)                            [FUNCTION]
;;;    Returns the value associated with a particular context using the
;;;    *version- ~ graph*. If context is illegal throws to an :error tag.
;;;       Like %get-value but does not require the object to have a
;;;    property $TYPE. object-id must not be a REF object, but a regular
;;;    object. Arguments:
;;;    obj-id: identifier of object
;;;    prop-id: identifier of local property
;;;    context: context
;;;    Return: the value associated with prop-id.
;;;
;;; %%GET-VALUE-AND-CONTEXT (obj-id prop-id)                        [FUNCTION]
;;;    Done to obtain value for objects which must not be versioned, like
;;;    counters ~ for entities.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    prop-id: local property identifier.
;;;    Retrurn:
;;;    a list whose car is context and cadr is the value.
;;;
;;; %GET-VALUE-FROM-CLASS-REF (class-ref expr)                      [FUNCTION]
;;;    returns the value associated with class-ref from a MOSS pattern.
;;;    Arguments:
;;;    class-ref: e.g. "address"
;;;    Returns:
;;;    value or nil.
;;;
;;; %GET-VALUE-FROM-PROP-REF (prop-ref expr                         [FUNCTION]
;;;                           &key (package *package*))
;;;    returns the value associated with prop-ref from a MOSS pattern.
;;;    Arguments:
;;;    prop-ref: e.g. "town"
;;;    package (key): package in which property has been defined
;;;    Returns:
;;;    value or nil.
;;;
;;; %GET-WORDS-FROM-TEXT (text delimiters)                          [FUNCTION]
;;;    takes an input string and a set of delimiters and returns a list of
;;;    words ~ separated by delimiters.
```

```
;;;    Arguments:
;;;    text: a string
;;;    delimiters: a set of characters that delimit a word
;;;    Return:
;;;    a list of words.
;;;
;;; %HAS-ID-LIST (entry inv-id class-id &key (context *context*))    [FUNCTION]
;;;    Returns the list of all objects ~
;;;         corresponding to entry point entry with inverse terminal
;;;    property inv-id ~ and of class class-id.
;;;       E.g. (%has-id-list 'ATTRIBUTE '$PNAM.OF '$ENT) returns ($EPT) ~
;;;         class-id must be specified locally; i.e., will not return
;;;    objects ~ that are instances of subclasses of class-id.
;;;       If context is illegal throws to an :error tag
;;;    Arguments:
;;;    entry: entry symbol
;;;    inv-id: local inverse attribute
;;;    class-id: identifier of target class
;;;    context: context
;;;    Return:
;;;    a list of objects.
;;;
;;; %%HAS-INVERSE-PROPERTIES (obj-id context)                        [FUNCTION]
;;;    Returns the list of inverse properties local to an object and having
;;;    ~ associated values in the specified context
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-PROPERTIES (obj-id context)                                [FUNCTION]
;;;    Returns the list of properties local to an object and having
;;;    associated ~ values in the specified context, looking directly at the
;;;    internal format ~ of the object. $TYPE is removed.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-PROPERTIES+ (obj-id context)                               [FUNCTION]
;;;    Returns the list of ALL properties LOCAL to an object even with no ~
;;;         value, in the specified context, looking directly at the
;;;    internal format ~ of the object.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
```

```
;;;
;;; %%HAS-STRUCTURAL-PROPERTIES (obj-id context)                      [FUNCTION]
;;;    Returns the list of structural properties local to an object and
;;;    having  ~ associated values in the specified context. $TYPE and $ID
;;;    are removed. Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-TERMINAL-PROPERTIES (obj-id context)                        [FUNCTION]
;;;    Returns the list of terminal properties local to an object and having
;;;    ~ associated values in the specified context.
;;;    Arguments:
;;;    obj-id: object identifier
;;;    context: context
;;;    Return:
;;;    a list of properties or nil.
;;;
;;; %%HAS-VALUE (obj-id prop-id context &key no-resolve)              [FUNCTION]
;;;    Returns the value list associated to a particular context if recorded
;;;    locally. No checks are done, except for legal context.
;;;    Arguments:
;;;    obj-id: object or ref to object under consideration (can be dead)
;;;    prop-id: property (may be local or inherited)
;;;    context: context (default current)
;;;    no-resolve (key): if true, skip the call to resolve intended to
;;;    solve multiclass belonging
;;;    Return:
;;;    the local value if any.
;;;
;;; INSERT-BEFORE (val val-l &rest val-test)                          [FUNCTION]
;;;    Should be a primitive.
;;;         Inserts item into a list either at the end of the list, or in
;;;    front of the ~ specified test-item, possibly duplicating it.
;;;    Arguments:
;;;    val: value to insert
;;;    val-l: list into which to insert the value
;;;    val-test (&rest): value that will be checked for insertion (equal
;;;    test) Return:
;;;       modified list or list of the value when val-l was nil
;;;
;;; INSERT-NTH (val val-l nth)                                        [FUNCTION]
;;;    Should be a primitive.
;;;         Inserts an item into a list at the nth position. If the list is
;;;    too short ~ then the value is inserted at the end of it.
;;;    Arguments:
;;;    val: value to be inserted
;;;    val-l: list into which to insert the value
;;;    nth: position at which the value must be inserted (0 is in front)
```

```
;;;    Return:
;;;    modified list
;;;
;;; %INTERSECT-SYMBOL-LISTS (ll1 ll2)                               [FUNCTION]
;;;    Intersect two lists of symbols by using their p-list. Done in linear
;;;    time.
;;;
;;; %INVERSE-PROPERTY-ID (prop-id &key (context *context*))         [FUNCTION]
;;;    Computes the inverse property id of a given structural or terminal
;;;    property Optional arg: version. When the property is still undefined,
;;;    calls %make-id-for-inverse-property.
;;;    For inverse properties of an inverse property return a list if
;;;    more than one, ~ e.g. when we have a property lattice.
;;;    Arguments:
;;;    prop-id: identifier of property
;;;    context (opt): context default current
;;;    Returns:
;;;    an inverse property identifier
;;;
;;; %IS-A? (obj1 obj2 &key (context *context*))                     [FUNCTION]
;;;    Checks if obj1 is equal to obj2, or obj-2 belongs to the transitive
;;;    closure of obj1~ using property $IS-A.
;;;    Arguments:
;;;    obj1: first object
;;;    obj2: second object
;;;    context (opt): context (default current)
;;;
;;; %IS-ATTRIBUTE? (prop-id &key (context *context*))               [FUNCTION]
;;;    Checks if the identifier points towards an attribute
;;;    Arguments:
;;;    prop-id: object identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    T or nil
;;;
;;; %IS-ATTRIBUTE-MODEL? (object-id &key (context *context*))       [FUNCTION]
;;;    Checks if the object is $EPT or some sub-type.
;;;    Arguments:
;;;    object-id: object identifier
;;;    context (opt): context (default current)
;;;    Return:
;;;    T or nil
;;;
;;; %IS-CLASS? (object-id &key (context *context*))                 [FUNCTION]
;;;    Checks if the object is a class, i.e. if its type is $ENT or a
;;;    subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
```

```
;;;
;;; %IS-CONCEPT? (object-id &key (context *context*))                [FUNCTION]
;;;    Checks if the object is a concept (same as class), i.e. if its type
;;;    is $ENT or a subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-COUNTER-MODEL? (object-id &key (context *context*))          [FUNCTION]
;;;    Checks if the object is $CTR or some sub-type.
;;;    Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %IS-ENTITY-MODEL? (object-id &key (context *context*))           [FUNCTION]
;;;    Checks if the object is $ENT or some sub-type.
;;;
;;; %IS-ENTRY-POINT-INSTANCE? (object-id &key (context *context*))   [FUNCTION]
;;;    Checks if the object is an instance of entry-point.
;;;
;;; %IS-GENERIC-PROPERTY-ID? (object-id &key (context *context*))    [FUNCTION]
;;;    Checks if the object is $EPR and is not a subproperty of some other
;;;    property.
;;;
;;; %IS-INSTANCE-OF? (object-id class-id)                            [FUNCTION]
;;;    Checks if the object is an instance of class id or of one of its
;;;    children. Arguments:
;;;    object-id: id of the object too test
;;;    class-id: id of the class
;;;    Return:
;;;    t if true nil otherwise.
;;;
;;; %IS-INVERSE-LINK-MODEL? (object-id &key (context *context*))     [FUNCTION]
;;;    Checks if the object is $EIL or some sub-type.
;;;
;;; %IS-INVERSE-PROPERTY? (xx &key (context *context*))              [FUNCTION]
;;;    Checks if the id points towards an inverse property, or to a property
;;;    containing $EIL in its transitive closure along $IS-A.
;;;
;;; %IS-INVERSE-PROPERTY-REF? (xx &key (context *context*))          [FUNCTION]
;;;    Checks if the string is the reference of an inverse property, i.e. a
;;;    string ~ starting with the > symbol whose name is that of a property.
;;;    Argument:
;;;    xx: string
;;;    Return:
;;;    the inverse property-id or nil.
;;;
```

```
;;; %IS-METHOD? (xx &key (context *context*))                     [FUNCTION]
;;;    Checks if the id points towards a method - Optional arg: context
;;;
;;; %IS-METHOD-MODEL? (object-id &key (context *context*))         [FUNCTION]
;;;    Checks if the object is $EFN or some sub-type.
;;;
;;; %IS-MODEL? (object-id &key (context *context*))                [FUNCTION]
;;;    Checks whether an object is a model (class) or not - Models ~
;;;         are objects that have properties like $PT or $PS or $RDX or
;;;    $CTRS ~ or whose type is in the transitive closure of $ENT along the
;;;    $IS-A.OF link.
;;;
;;; %IS-ORPHAN? (object-id &key (context *context*))               [FUNCTION]
;;;    Short synonym for %classless-object?
;;;
;;; %IS-RELATION? (prop-id &key (context *context*))               [FUNCTION]
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-RELATIONSHIP? (prop-id &key (context *context*))           [FUNCTION]
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-RELATION-MODEL? (object-id &key (context *context*))       [FUNCTION]
;;;    checks if the object is $EPS or some sub-type
;;;
;;; %IS-STRUCTURAL-PROPERTY? (prop-id &key (context *context*))    [FUNCTION]
;;;    Checks if the identifier points towards a structural property -
;;;     Optional arg: context
;;;
;;; %IS-STRUCTURAL-PROPERTY-MODEL? (object-id                      [FUNCTION]
;;;                                 &key (context *context*))
;;;    Checks if the object is $EPS or some sub-type.
;;;
;;; %IS-SYSTEM? (object-id &key (context *context*))               [FUNCTION]
;;;    Checks if the id points towards a system - Optional arg: context
;;;
;;; %IS-SYSTEM-MODEL? (object-id &key (context *context*))         [FUNCTION]
;;;    Checks if the entity is a model, and belongs to the MOSS package.~
;;;         Arguments:
;;;    object-id: identifier of object to test
;;;    context (opt): context default current
;;;    Return:
;;;    nil or t
;;;
;;; %IS-TERMINAL-PROPERTY? (prop-id &key (context *context*))      [FUNCTION]
;;;    Checks if the identifier points towards a terminal property -
;;;     Optional arg: context
;;;
;;; %IS-TERMINAL-PROPERTY-MODEL? (object-id                        [FUNCTION]
;;;                               &key (context *context*))
;;;    Checks if the object is $EPT or some sub-type.
```

```
;;;
;;; %IS-UNIVERSAL-METHOD? (xx &key (context *context*))             [FUNCTION]
;;;    Checks if the id points towards a universal method - Optional arg:
;;;    context
;;;
;;; %IS-UNIVERSAL-METHOD-MODEL? (object-id                          [FUNCTION]
;;;                              &key (context *context*))
;;;    Checks if the object is $UNI or some sub-type.
;;;
;;; %IS-VALUE-TYPE? (value value-type)                              [FUNCTION]
;;;    Checks that a value has the right type.
;;;    Arguments:
;;;    value: value to check
;;;    value-type: one of the XSD types (birkkkhh!)
;;;    Returns:
;;;    value if the right type, nil otherwise.
;;;
;;; %IS-VARIABLE-NAME? (ref)                                        [FUNCTION]
;;;    Checks if the refernce is a variable name, i.e. a symbol starting
;;;    with _ (underscore)
;;;
;;; %IS-VIRTUAL-CONCEPT? (object-id &key (context *context*))       [FUNCTION]
;;;    Checks if the object is a concept (same as class), i.e. if its type
;;;    is $ENT or a subtype of $ENT. Arguments:
;;;    object-id: identifier of the object to check
;;;    context (opt): context default current
;;;    Return
;;;    nil or a list result of an intersection.
;;;
;;; %LINK (obj1-id sp-id obj2-id &key (context *context*))          [FUNCTION]
;;;    Brute force link two objects using the structural property id - ~
;;;         No cardinality check is done - sp-id must be direct property.
;;;       (%link obj1-id sp-id obj2-id context) - links two objects ~
;;;         using sp-id - no checking done on args - crude link to be able
;;;    ~ to bootstrap without the properties really existing. Here we
;;;    manipulate ~ the names. We assume that sp-id is not an inverse
;;;    property ~ If the link already exists then does nothing. Otherwise,
;;;    introduces a ~ new link modifying or creating a value in the
;;;    specified context. ~ This might look like a bit complicated a piece
;;;    of code, however we try ~ to do simple tests first to avoid long
;;;    execution times. Arguments:
;;;    obj1-id: identifier for first object
;;;    sp-id: identifier of the local linking property
;;;    obj2-id: identifier for the second object
;;;    context: context
;;;    Return:
;;;    list representing the first object
;;;
;;; %LINK-ALL (obj1-id sp-id obj-list &key (context *context*))     [FUNCTION]
;;;    Brute force link an object to a list of objects using the structural
```

```
;;;    property ~ id - No cardinality check is done - sp-id must be direct
;;;    property. (%link obj1-id sp-id obj-list context) - links two objects
;;;    ~ using sp-id - no checking done on args - crude link to be able ~
;;;         to bootstrap without the properties really existing. Here we
;;;    manipulate ~ the names. We assume that sp-id is not an inverse
;;;    property ~ If the link already exists then does nothing. Otherwise,
;;;    introduces a ~ new link modifying or creating a value in the
;;;    specified context. ~ This might look like a bit complicated a piece
;;;    of code, however we try ~ to do simple tests first to avoid long
;;;    execution times. Arguments:
;;;    obj1-id: identifier for first object
;;;    sp-id: identifier of the local linking property
;;;    obj-list: identifier for the second object
;;;    context: context
;;;    Return:
;;;    list representing the first object
;;;
;;; %MAKE-ENTITY-SUBTREE (ens &key (context *context*))             [FUNCTION]
;;;    takes an entity and builds the subtree of its sub-classes, e.g.
;;;    (A (B C (D E) F))
;;;    where (D E) are children of C and (B C F) are children of A.
;;;    Arguments:
;;;    ens: entity to process
;;;    Return:
;;;    (ens) if no subtree, or (ens <subtree>).
;;;
;;; %MAKE-ENTITY-TREE (&key (context *context*))                    [FUNCTION]
;;;    builds the forest corresponfing to the application classes.
;;;    Arguments:
;;;    context (opt): default is *context*
;;;    Return:
;;;    a tree like (A (B C (D E) F) G H)
;;;    where (D E) are children of C and (B C F) are children of A, and G
;;;    and H are parallel trees.
;;;
;;; %MAKE-ENTITY-TREE (&key (context *context*)                     [FUNCTION]
;;;                     (package *package*))
;;;    builds the forest keeping classes from the current package.
;;;    Arguments:
;;;    context (opt): default is *context*
;;;    Return:
;;;    a tree like (A (B C (D E) F) G H)
;;;    where (D E) are children of C and (B C F) are children of A, and G
;;;    and H are parallel trees.
;;;
;;; %MAKE-ENTRY-SYMBOLS (multilingual-name &key type prefix         [FUNCTION]
;;;                       (package *package*))
;;;    Takes a multilingual-name as input and builds entry symbols for each
;;;    of the ~ synonyms in each specified language.
;;;    Note that a list of symbols is not a valid input argument.
```

```
;;;    Arguments:
;;;    multilingual-name: string, symbol or multilingual-name specifying
;;;    the entry points tp-id: local attibute identifier
;;;    type (key): type of MOSS object (default is nil)
;;;    prefix (key): prefix for produced symbols
;;;    Return:
;;;    list of the entry point symbols
;;;
;;; %MAKE-EP (entry tp-id obj-id &key (context *context*) export)    [FUNCTION]
;;;    Adds a new object to an entry-point if it exists - Otherwise creates
;;;    the entry point(s) and record it/them in the current system
;;;    (*moss-system*). Export the entry symbol(s).
;;;    Arguments:
;;;    entry: symbol or list of symbols specifying the entry point(s)
;;;    tp-id: local attibute identifier
;;;    obj-id: identifier of object to index
;;;    context (opt): context (default current)
;;;    export (key): it t export entry points from the package
;;;    Return:
;;;    internal format of the entry point object or a list of them
;;;
;;; %MAKE-EP-FROM-MLN (mln tp-id obj-id &key type                    [FUNCTION]
;;;                    (context *context*) export)
;;;    Takes a multilingual-name as input and builds entries for each of the
;;;    ~ synonyms in each specified LEGAL language in package of obj-id.
;;;    Arguments:
;;;    mln: legal multilingual-name specifying the entry points
;;;    tp-id: local attibute identifier
;;;    obj-id: identifier of object to index
;;;    context (opt): context (default current)
;;;    type (key): type of object, e.g. $ENT, $EPR, $FN, ... (see
;;;    %make-name) export (key): if t then we should export entry points
;;;    from their package Return:
;;;    list of the entry point object (internal format)
;;;
;;; %MAKE-ID (class-id &key name prefix value context id ideal       [FUNCTION]
;;;          (package *package*))
;;;    Generic function to make object identifiers. If the symbol exists,
;;;    then ~ we throw to :error.
;;;    Arguments:
;;;    class-id: identifier of the class of the object (*none* is
;;;    authorized) name (key): useful name, e.g.class  name or property name
;;;    prefix (key): additional name (string), e.g. for defining
;;;    properties (the class name) value (key): value for an instance
;;;    context (key): context default current
;;;    id (key): id of class or property
;;;    ideal (key): if there indicates we want an ideal, :id option must
;;;    be there package (key): package into which new symbol must be
;;;    inserted (default *package*) Return:
;;;    unique new symbol.
```

```
;;;
;;; %MAKE-INVERSE-PROPERTY (id multilingual-name                    [FUNCTION]
;;;                            &key (context *context*) export)
;;;    create an inverse property for an attribute or a relation. Links it
;;;    to the ~ direct property, creates the name and entry point, add it to
;;;    MOSS. It is created in the same package as id and in the specified
;;;    context. Uses the default language.
;;;    Arguments:
;;;    id: identifier of the property to invert
;;;    multilingual-name: name of the property to invert
;;;    Return:
;;;    :done
;;;
;;; %MAKE-INVERSE-PROPERTY-MLN (mln)                                [FUNCTION]
;;;    build an inverse multilingual name for the inverse property.
;;;    Arguments:
;;;    mln: multilingual name
;;;    Result:
;;;    a list of strings and an inverse mln
;;;
;;; %MAKE-NAME (class-id &rest option-list &key name                [FUNCTION]
;;;             (package *package*) &allow-other-keys)
;;;    Makes a name with %make-name-string and interns is in the specified
;;;    package. Arguments:
;;;    class-id: identifier of the corresponding class
;;;    package (key): package for interning symbol (default current
;;;    execution package) name (key): string, symbol or multilingual string
;;;     more keys are allowed as specified in the lambda-list of the
;;;    %make-string function Return:
;;;    interned symbol.
;;;
;;; %MAKE-NAME-STRING (class-id &key name method-class-id           [FUNCTION]
;;;                    method-type state-type short-name prefix
;;;                    (type "") (context *context*)
;;;                    &allow-other-keys &aux pfx)
;;;    Builds name strings for various moss entities, e.g. method names,
;;;    inverse-property ~ names, internal method function names, etc.
;;;    Examples of resulting strings
;;;        XXX-Y-ZZZ from " Xxx y'zzz "
;;;        IS-XXX-OF inverse property
;;;        HAS-XXX  property
;;;        $E-PERSON=S=0=PRINT-SELF own method internal function name
;;;        $E-PERSON=I=0=SUMMARY instance method internal function name
;;;       *0=PRINT-OBJECT universal method internal function name
;;;        $-PER typeless radix
;;;        _HAS-BROTHER  internal variable
;;;    We mix English prefix and suffix with other languages, which is
;;;    not crucial ~ since such names are internal to MOSS.
;;;    Arguments:
;;;    class-id: identifier of the corresponding class
```

```
;;;    name (key): name, e.g. of method
;;;    context (key): context reference
;;;    prefix (key): symbol, string or mln specifying a class (for class
;;;    properties) method-type (key): instance or own or universal
;;;    method-class-id (key): class-id for a given method
;;;    state-type: for state objects {:entry-state, :success, :failure}
;;;    short-name: for state objects, prefix
;;;    package (key): package for interning symbol (default current
;;;    execution package) context (key): context used for building some
;;;    names (default current) language (key): language to use, e.g. :en,
;;;    :fr (default *language*) Return:
;;;    a single name string
;;;
;;; %MAKE-PHRASE (&rest words)                                  [FUNCTION]
;;;    Takes a list of words and returns a string with the words separated
;;;    by a space. Arguments:
;;;    words (rest): strings
;;;    Return:
;;;    a string.
;;;
;;; %MAKE-RADIX-FROM-NAME (name &key (type "") (package *package*))  [FUNCTION]
;;;    build a radix name for a property or for a class. Tries to build a
;;;    unique name ~ using the first 3 letters of the name. Error if the
;;;    symbol already exists. Arguments:
;;;    name: e.g. "SIGLE"
;;;    type (opt): e.g. "T " (for terminal property)
;;;    Return:
;;;    e.g. $T-SIG or error if already exists
;;;
;;; %MAKE-REF-OBJECT (class-ref obj-id &key (context *context*))     [FUNCTION]
;;;    Builds a reference object, checking the validity of the class. If
;;;    invalid ~ throws to :error tag.
;;;    Arguments:
;;;    class-ref: reference of the new class
;;;    obj-id: id of object being referenced
;;;    context (opt): context, default current
;;;    Return:
;;;    id of new ref object.
;;;
;;; %MAKE-STRING-FROM-PATTERN (control-string pattern               [FUNCTION]
;;;                            &rest properties)
;;;    makes a string from a pattern (XML like). The first element of
;;;    pattern is the ~ class of the  object. The function ~
;;;    is somewhat equivalent to the =summary method, but the method is
;;;    applied to ~ the structured object.
;;;    Ex: (fn "~{~A~^ ~}, ~{~A~^ ~}"
;;;              '("person" ("name" "Dupond") ("first-name" "Jean"))
;;;              "name" "first-name")
;;;    returns "Dupond, Jean"
;;;    Arguments:
```

```
;;;    pattern: the pattern
;;;    data: the data
;;;    Return:
;;;    string to be printed.
;;;
;;; %MAKE-WORD-COMBINATIONS (word-list)                              [FUNCTION]
;;;
;;; %MAKE-WORD-LIST (text &key (norm t))                             [FUNCTION]
;;;    Norms a text string by separating each word making it lower case with
;;;    a leading~ uppercased letter.
;;;    Arguments:
;;;    text: text string
;;;    norm (key): if true (default) capitalize the first letter of each
;;;    word Return:
;;;    a list of normed words.
;;;
;;; %%MERGE-OBJECTS (obj-id obj-list context)                        [FUNCTION]
;;;    obj-id is the id of an object already existing in memory when we are
;;;    loading a new object with same id from disk. The new format of the
;;;    loaded object is obj-list. We compare properties of the existing
;;;    (moss) object and the loaded object and update all properties that
;;;    are not system or that contain application references. For a tp, the
;;;    system object is the reference. The result is a merged object
;;;    including system and application data. Arguments:
;;;    obj-id: id of the object to be processed
;;;    obj-list: object saved by the application
;;;    Returns:
;;;    the list representing the system object merged with application
;;;    references.
;;;
;;; %MLN? (expr)                                                     [FUNCTION]
;;;    Checks whether a list is a multilingual name Uses global
;;;    *language-tags* variable. ~ nil is not a valid MLN.
;;;    Argument:
;;;    expr: something like (:en "London" :fr "Londres") or (:name :en
;;;    "London") Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-1? (expr)                                                   [FUNCTION]
;;;    Checks whether a list is a multilingual string. Uses global
;;;    *language-tags* variable. Argument:
;;;    expr: something like (:en "London" :fr "Londres")
;;;    Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-ADD-VALUE (mln value language-tag)                          [FUNCTION]
;;;    Adds a value corresponding to a specific language at the end of the
;;;    list. ~ Does not check if value is already there.
;;;    Arguments:
;;;    mln: multilingual name
```

```
;;;    value: coerced to a string
;;;    language-tag: must be legal, i.e. part of *language-tags*
;;;    Return:
;;;    the modified mln.
;;;
;;; %MLN-EQUAL (mln1 mln2 &key language)                             [FUNCTION]
;;;    Checks whether two multilingual-names are equal. They may also be
;;;    strings. ~ When the language key argument is not there uses global
;;;    *language* variable.~ When *language* is unbound, tries all possible
;;;    languages. The presence of ~ language is only necessary if one of the
;;;    arguments is a string. The function uses %string-norm before
;;;    comparing strings. Argument:
;;;    mln1: a multilingual name or a string
;;;    mln2: id.
;;;    language (key): a specific language tag
;;;    Result:
;;;    T if OK, nil otherwise
;;;
;;; %MLN-EXTRACT-ALL-SYNONYMS (mln)                                  [FUNCTION]
;;;    Extracts all synonyms as a list of strings regardless of the
;;;    language. Arguments:
;;;    mln: multilingual name
;;;    Return
;;;    a list of strings.
;;;
;;; %MLN-EXTRACT-MLN-FROM-REF (ref language                          [FUNCTION]
;;;                            &key no-throw-on-error (default :en)
;;;                            &aux $$$)
;;;    takes a ref input symbol string or mln and builds a specific mln
;;;    restricted to ~ the specified language. If language is :all and ref
;;;    is a string or symbol, then ~ language defaults to default. If ref is
;;;    an MLN, then language defaults to default ~ if there, first found
;;;    language otherwise (canonical behavior). Default beavior ~ is enabled
;;;    only if no-throw-on-error option is set to T. Removes also the
;;;    leading :name marker if present in mln. Arguments:
;;;    ref: a string symbol or mln
;;;    language a language tag or :all
;;;    no-throw (key): if true function does not throw on error but
;;;    returns nil default (key): default language tag (default: :EN)
;;;    Result:
;;;    a valid mln or throws to :error
;;;
;;; %MLN-FILTER-LANGUAGE (mln language-tag &key always)              [FUNCTION]
;;;    Extracts from the MLN the synonym string corresponding to specified
;;;    language. ~ If mln is a string or language is :all, then returns mln
;;;    untouched, unless always ~ is true in which case returns the English
;;;    terms if there, otherwise random terms. If language is not legal,
;;;    throws to :error. Arguments:
;;;    mln: a multilingual name (can also be a simple string)
;;;    language-tag: a legal language tag (part of *language-tags*)
```

```
;;;    always (key): if t then returns either the English terms or a
;;;    random one Return:
;;;    a string or nil if none.
;;;
;;; %MLN-GET-CANONICAL-NAME (mln &aux names)                    [FUNCTION]
;;;    Extracts from a multilingual name the canonical name that will be
;;;    used to build ~ an object ID. By default it is the first name
;;;    corresponding to the value of *language*, ~ or else the first name of
;;;    the English entry, or else the name of the list. ~ An error occurs
;;;    when the argument is not multilingual name. Arguments:
;;;    mln: a multilingual name
;;;    Return:
;;;    2 values: a simple string (presumably for building an ID) and the
;;;    language tag. Error:
;;;    error if not a multilingual name.
;;;
;;; %MLN-GET-FIRST-NAME (name-string)                           [FUNCTION]
;;;    Extracts the first name from the name string. Names are separated by
;;;    semi-columns. E.g. "identity card ; papers" returns "identity card "
;;;    Argument:
;;;    name-string
;;;    Return:
;;;    string with the first name.
;;;
;;; %MLN-MEMBER (input-string tag mln)                          [FUNCTION]
;;;    checks whether the input string is one of the synonyms of the mln in
;;;    the ~ language specified by tag. If tag is :all then we check against
;;;    any synonym ~ in any language.
;;;    Arguments:
;;;    input-string: input string (to be normed with %norm-string)
;;;    tag; a legal language tag or :all
;;;    mln: a mltilingual name
;;;    Return:
;;;    non nil value if true.
;;;
;;; %MLN-MERGE (&rest mln)                                      [FUNCTION]
;;;    Takes a list of MLN and produces a single MLN as a result, merging
;;;    the ~ synonyms (using or norm-string comparison) and keeping the
;;;    order of the ~ inputs.
;;;    Arguments:
;;;    mln: a multilingual name
;;;    more-mln (rest): more multilingual names
;;;    Return:
;;;    a single multilingual name
;;;    Error:
;;;    in case one of the mln has a bad format.
;;;
;;; %MLN-NORM (expr)                                            [FUNCTION]
;;;    Norms an mln expr by removing the :name keyword if there.
;;;    Argument:
```

```
;;;    expr: expression to test
;;;    Return:
;;;    normed expr, nil is an error)
;;;
;;; %MLN-PRINT-STRING (mln &optional (language-tag :all))           [FUNCTION]
;;;    Prints a multilingual name nicely.
;;;    Arguments:
;;;    mln: multilingual name
;;;    language-tag: legal language-tag (default :all)
;;;    Result:
;;;    a string ready to be printed (can be empty).
;;;
;;; %MLN-REMOVE-LANGUAGE (mln language-tag)                         [FUNCTION]
;;;    Removes the entry corresponding to a language from a multilingual
;;;    name. Arguments:
;;;    mln: multilanguage name
;;;    language-tag: language to remove (must be legal
;;;    Return:
;;;    modified mln.
;;;
;;; %MLN-REMOVE-VALUE (mln value language-tag)                      [FUNCTION]
;;;    Removes a value in the list of synonyms of a particular language.
;;;    Arguments:
;;;    mln: multilingual name
;;;    value: coerced to a string (loose package context)
;;;    language-tag: must be legal
;;;    Return:
;;;    the modified mln.
;;;
;;; %MLN-SET-VALUE (mln language-tag syn-string)                    [FUNCTION]
;;;    Sets the language part to the specified value, erasing the old value.
;;;    Arguments:
;;;    mln: a multilingual name
;;;    language-tag: language tag
;;;    syn-string: coerced to a string (loose package context)
;;;    Return:
;;;    the modified normed MLN.
;;;
;;; %MULTILINGUAL-NAME? (expr)                                      [FUNCTION]
;;;    Checks whether a list is a multilingual string. Uses global
;;;    *language-tags* variable. Argument:
;;;    expr: something like (:en "London" :fr "Londres")
;;;    Result:
;;;    T if OK, nil otherwise.
;;;
;;; %PCLASS? (input class-ref)                                      [FUNCTION]
;;;    tests if input belongs to the transitive closure of class-ref. Only
;;;    works in ~ if the language is that if the input and class ref.
;;;    Arguments:
;;;    input: some expression that must be a string
```

```
;;;    class-ref: a string naming a class
;;;    Return:
;;;    nil if not the case, something not nil otherwise.
;;;
;;; %PDM? (xx)                                                  [FUNCTION]
;;;    Checks if an object had the right PDM format, i.e. it must be a
;;;    symbol, bound, ~ be an alist, and have a local $TYPE property.
;;;    References are PDM objects
;;;
;;; %PEP (obj-id &key (version *context*) (offset 30) (stream t))    [FUNCTION]
;;;    Prints versions of values from a given context up to the root.
;;;       (%pep obj-id version &rest offset)
;;;       For each property prints values in all previous context up to the
;;;    root ~ does not check for illegal context since it is a printing
;;;    function.
;;;
;;; %PEP-PV (value context-branch &optional (stream t) offset)      [FUNCTION]
;;;    Used by %pep, prints ~
;;;         a set of value associated with a property from the root to
;;;    current context. Offset is currently set to 30.
;;;
;;; %PFORMAT (fstring input &rest prop-list)                    [FUNCTION]
;;;    produces a string for printing from a list produced typically by the
;;;    ~ =make-print-string method.
;;;    Arguments:
;;;    input: a-list, e.g. (("name" "Dupond")("first name" "Jean")...)
;;;    fstring: string control format
;;;    prop-list: a list of properties appearing in the a-list. last
;;;    entry can be the specification of a particular package
;;;    Return:
;;;    a string: e.g. "Dupond, Jean"
;;;
;;; %PUTC (obj-id value prop-id context)                        [FUNCTION]
;;;    Stores a versioned value onto the p-list of the object used as a
;;;    cache.
;;;
;;; %PUTM (object-id function-name method-name context          [FUNCTION]
;;;        &rest own-or-instance)
;;;    Records the function name corresponding to the method onto the ~
;;;         p-list of the object. Own-or-instance can be either one of the
;;;    ~ keywords :own or :instance; If not present, then :general is used.
;;;
;;; %RANK-ENTITIES-WRT-WORD-LIST (entity-list word-list)        [FUNCTION]
;;;    takes a list of entity ids and a list of words, and returns a score
;;;    for ~ each entity that is function of the number of apparitions of
;;;    the words in ~ attributes of the entity.
;;;    Arguments:
;;;    entity-list: a list of object ids (ei)
;;;    word-list: a list of words (wordj)
;;;    Returns:
```

```
;;;    a list of pairs (ei wi), e.g. (($e-person.2 0.75)($E-PERSON.3
;;;    0.5)).
;;;
;;; %REMOVE-REDUNDANT-PROPERTIES (prop-list                          [FUNCTION]
;;;                              &key (context *context*))
;;;    Removes properties that share the same generic property and keep the
;;;    leftmost ~ one.
;;;    Arguments:
;;;    prop-list: list of property ids
;;;    context (opt): context (default current)
;;;    Return:
;;;    a cleaned list of properties.
;;;
;;; %%REMPROP (obj-id prop-id context)                               [FUNCTION]
;;;    Removes the values associated with ~
;;;         the specified property in the specified context. The net result
;;;    is ~ that the object inherits the values from a higher level.
;;;       So the resulting value in the context will probably not be nil.
;;;    Hence ~ it is not equivalent to a set-value with a nil argument.
;;;
;;; %%REMVAL (obj-id val prop-id context)                            [FUNCTION]
;;;    Removes the value associated with ~
;;;         a property in a given context. Normally useful for terminal
;;;    properties ~ However does not check for entry-points. Also val must
;;;    be a normalized value n i.e. must be expressed using the internal
;;;    format (it cannot be an external value).
;;;
;;; %%REMNTHVAL (obj-id val prop-id context position)                [FUNCTION]
;;;    Removes the nth value associated ~
;;;         with a property in a given context. Normally useful for
;;;    terminal properties ~ However does not check for entry-points. Does
;;;    not return any significant value.
;;;
;;; %RESET (&key (context *context* there?))                         [FUNCTION]
;;;    Removes all classes and instances of an application, making the
;;;    corresponding ~ symbols unbound. Protects all data defined in the
;;;    kernel (moss  package). Arguments:
;;;    none
;;;    Return:
;;;    t
;;;
;;; %RESOLVE (id &key (context *context*))                           [FUNCTION]
;;;    Replaces id with the actual id when it is a ref object. Otherwise
;;;    returns id. When id is unbound returns id.
;;;    Arguments:
;;;    id: object id or REF identifier
;;;    Return:
;;;    resolved id.
;;;
;;; %SELECT-BEST-ENTITIES (entity-score-list)                        [FUNCTION]
```

```
;;;    select entities with highest score and return the list.
;;;    Arguments:
;;;    entity-score-list: a list like (($E-PERSON.2 0.5)(..)...)
;;;    Returns:
;;;    the list of entities with the highest score.
;;;
;;; %%SET-SYMBOL-PACKAGE (symbol package)                         [FUNCTION]
;;;    if package is nil set it to default *package*, and create symbol in
;;;    the ~ specified package. No check on input.
;;;    Arguments:
;;;    symbol: any symbol
;;;    package: a valid package or keyword
;;;    Return:
;;;    a symbol interned in the package.
;;;
;;; %%SET-VALUE (obj-id value prop-id context)                    [FUNCTION]
;;;    Resets the value associated ~
;;;         with a property in a given context. Replaces previous values.
;;;    Does not ~ try to inherit using the version-graph, but defines a new
;;;    context locally ~ Thus, it is not an %add-value, but an actual
;;;    %%set-value.
;;;
;;; %%SET-VALUE-LIST (obj-id value-list prop-id context)          [FUNCTION]
;;;    Resets the value associated ~
;;;         with a property in a given context. Replaces previous values.
;;;    Does not ~ try to inherit using the version-graph, but defines a new
;;;    context locally ~ Thus, it is not an %add-value, but an actual
;;;    %%set-value - ~ No entry point is created.
;;;
;;; %SP-GAMMA (node arc-label &key (context *context*))           [FUNCTION]
;;;    Returns the transitive closure ~
;;;         of objects for a given version.
;;;       This is the famous gamma function found in graph theory.
;;;       No check that arc-label is a structural property.
;;;    Arguments:
;;;    node: starting node
;;;    arc-label: label to follow on the arcs (e.g. $IS-A)
;;;    context (opt): context, default value is *context* (current)
;;;
;;; %SP-GAMMA-L (node-list arc-label &key (context *context*))    [FUNCTION]
;;;    Computes a transitive closure
;;;    Arguments:
;;;    node-list: a list of nodes from which to navigate
;;;    arc-label: the label of the arc we want to use
;;;    context (opt): context default current
;;;    Return:
;;;    a list of nodes including those of the node-list.
;;;
;;; %SP-GAMMA1 (ll prop stack old-set version)                    [FUNCTION]
;;;    Called by %sp-gamma for computing a transitive closure.
```

```
;;;       gamma1 does the job.
;;;       computes the whole stuff depth first using a stack of yet
;;;    unprocessed ~ candidates.
;;;     ll contains the list of nodes being processed
;;;     stack the ones that are waiting
;;;     old-set contains partial results.
;;;    Arguments:
;;;    ll: candidate list of nodes to be examined
;;;    prop: property along which we make the closure
;;;    stack: stack of waiting candidates
;;;    old-set: list containing the result
;;;    version: context
;;;    Return:
;;;    list of nodes making the transitive closure.
;;;
;;; STR-EQUAL (aa bb)                                         [FUNCTION]
;;;
;;; %STRING-NORM (input &optional (interchar #\-))            [FUNCTION]
;;;    Same as make-value-string but with more explicit name.
;;;    If input is a multilingual string, uses the canonical name.
;;;    If input is NIL throws to :error
;;;    Arguments:
;;;    input: may be a string, a symbol or a multilingual string
;;;    interchar: character to hyphenate the final string (defaul is #-
;;;    Return:
;;;    a normed string, e.g. "AU-JOUR-D-AUJOURD-HUI ".
;;;
;;; %SYNONYM-ADD (syn-string value)                           [FUNCTION]
;;;    Adds a value to a string at the end.
;;;    Sends a warning if language tag does not exist and does not add
;;;    value. Arguments:
;;;    syn-string: a synonym string or nil
;;;    value: value to be included (coerced to string)
;;;    Return:
;;;    modified string when no error, original string otherwise.
;;;
;;; %SYNONYM-EXPLODE (text)                                   [FUNCTION]
;;;    Takes a string, consider it as a synonym string and extract items
;;;    separated ~ by a semi-column. Returns the list of string items.
;;;    Arguments:
;;;    text: a string
;;;    Return
;;;    a list of strings.
;;;
;;; %SYNONYM-MAKE (&rest item-list)                           [FUNCTION]
;;;    Builds a synonym string with a list of items . Eadh item is coerced
;;;    to a string. Arguments:
;;;    item-list: a list of items
;;;    Return:
;;;    a synonym string.
```

```
;;;
;;; %SYNONYM-MEMBER (value text)                                [FUNCTION]
;;;    Checks if a string is part of the synonym list. Uses the %string-norm
;;;    ~ function to normalize the strings before comparing.
;;;    Arguments:
;;;    value: a value coerced to a string
;;;    text: a synonym string
;;;    Return:
;;;    a list of unnormed remaining synonyms if succcess, NIL if nothing
;;;    left.
;;;
;;; %SYNONYM-MERGE-STRINGS (&rest names)                        [FUNCTION]
;;;    Merges several synonym string, removing duplicates (using a
;;;    norm-string ~ comparison) and preserving the order.
;;;    Arguments:
;;;    name: a possible complex string
;;;    more-names (rest): more of that
;;;    Return:
;;;    a single complex string
;;;    Error:
;;;    when some of the aguments are not strings.
;;;
;;; %SYNONYM-REMOVE (value text)                                [FUNCTION]
;;;    Removes a synonym from the list of synonyms. If nothing is left
;;;    return ~ empty string.
;;;    Arguments:
;;;    value: a value coerced to a string
;;;    text: a synonym string
;;;    Return:
;;;    the string with the value removed NIL if nothing left.
;;;
;;; %SUBTYPE? (object-id model-id &key (context *context*))     [FUNCTION]
;;;    Checks if object-id is a sub-type of model-id including itself.
;;;    Arguments:
;;;    object-id: identifier of object to check for modelhood
;;;    model-id: reference-model
;;;    context (opt): context default current
;;;    Return:
;;;    T or nil.
;;;
;;; %TYPE? (object-id class-id &key (context *context*))        [FUNCTION]
;;;    Checks if one of the classes of object-id is class-id or one of its
;;;    ~ subclasses.
;;;       If class is *any* then the result is true.
;;;    Arguments:
;;;    object-id: the id of a pdm object
;;;    class-id: the id of a class
;;;    Returns
;;;    nil or T.
;;;
```

```
;;; %TYPE-OF (obj-id &key (context *context*))                     [FUNCTION]
;;;    Extracts the type from an object (name of its class(es)).
;;;    Careful: returns "CONS" if following a programming error ~
;;;         obj-id pass the value of the identifier...
;;;    Argument:
;;;    obj-id: identifier of a particular object
;;;    Return:
;;;    a list containing the identifiers of the class(es) of which the
;;;    object is an instance ~ or its lisp type if it is not a PDM object.
;;;
;;; %UNLINK (obj1-id sp-id obj2-id &key (context *context*))        [FUNCTION]
;;;    Disconnects to objects by removing the structural link.
;;;         in the given context. If the context does not exist then one
;;;    has to ~ recover the states of the links in the required context.
;;;    Then if the ~ link did not exist, then one exits doing nothing.
;;;    Otherwise, one removes ~ the link and stores the result.
;;;
;;; %VALIDATE-SP (sp-id suc-list &key (context *context*))          [FUNCTION]
;;;    Checks if the property successors obey the restrictions attached to
;;;    the relation. Arguments:
;;;    sp-id: id of the relation
;;;    suc-list: list of successors to check
;;;    context (opt): context (default current)
;;;    Return:
;;;    2 values:
;;;          1st value value-list if OK, nil otherwise
;;;          2nd value: list of string error message.
;;;
;;; %VALIDATE-TP (tp-id value-list &key (context *context*))        [FUNCTION]
;;;    Checks if the property values obey the restrictions attached to the
;;;    attribute. Arguments:
;;;    tp-id: id of the attribute
;;;    value-list: list of values to check
;;;    context (opt): context (default current)
;;;    Return:
;;;    2 values:
;;;          1st value value-list if OK, nil otherwise
;;;          2nd value: list of string error message.
;;;
;;; %VALIDATE-TP-VALUE (restriction data &key prop-id obj-id        [FUNCTION]
;;;                     (context *context*))
;;;    Takes some data and checks if they agree with a value-restriction
;;;    condition,~ associated with a terminal property. Used by the =xi
;;;    method whenever ~ we want to put data as the value of the property.
;;;    Processed conditions are those of *allowed-restriction-operators*
;;;    Arguments:
;;;    restriction: restriction condition associated with the property,
;;;    e.g. (:one-of (1 2 3)) data: a list of values to be checked
;;;    prop-id (key): identifier of the local property (unused)
;;;     obj-id (key): identifier of the object being processed (unued)
```

```
;;;    context (key): context default current (unused)
;;;    Return:
;;;    data when validated, nil otherwise.
;;;
;;; %VG-GAMMA (context)                                                [FUNCTION]
;;;    Computes the transitive closure upwards on the version graph.
;;;       Get the transitive closure of the context i.e. a merge of all the
;;;    paths ~ from context to root without duplicates, traversed in a
;;;    breadth first ~ fashion, since we assume that the value should not be
;;;    far from our context ~ This list should be computed once for all when
;;;    the context is set ~ e.g. in a %set-context function
;;;    Argument:
;;;    context: starting context
;;;    Return:
;;;    a list of contexts.
;;;
;;; %VG-GAMMA-L (lres lnext ll pred)                                   [FUNCTION]
;;;    Service function used be %vg-gamma
;;;    Arguments:
;;;    lres: resulting list
;;;    lnext: next candidates to examine later
;;;    ll: working list of candidates
;;;    pred: list of future nodes to process
;;;    Return:
;;;    lres
;;;
;;; %%ZAP-OBJECT (obj-id)                                              [FUNCTION]
;;;    Deletes an object by removing all values from all contexts and making
;;;    its ~ identifier unbound. This can destroy the database consistency.
;;;    Arguments:
;;;    obj-id: identifier of the object
;;;    Return:
;;;    not significant.
;;;
;;; %ZAP-PLIST (obj-id)                                                [FUNCTION]
;;;    Clears the plist of a given symbol.
;;;
```

Chapter 9

Query System

Contents

9.1 Introduction

What is a query system in an object system?

A query system is a piece of software that allows to access objects according to some criteria, e.g. "all persons who have a red car." It is an essential part of any database or knowledge base system.

For some object systems the query system has to face a serious problem. Indeed, objects are commonly spread randomly across the disk area, and for very large object bases one needs to perform many disk reads; however, disk reads are expensive. It is therefore crucial to minimize the number of such disk reads by organizing the sequence of accesses cleverly. This was already the case for the OQA system processing all requests onto the VORAS prototype. The OQA system was improved by Li Yi (Li Yi 89), who introduced the possibility of having OR and NOT operators in a query, and processing the query without having to reduce the query logically to conjunctive normal form, nor to disjunctive normal form. Indeed, while doing so, one tends to increase the number of redundant disk reads.

The current version of the query system was changed in version 3.2 to improve the expressiveness of the allowable requests. The system was first upgraded to MOSS v4.2. This document describes the upgraded version to MOSS v7, which is not very different from the previous one.

Warning In this document we use indifferently the terms class or concepts, and individual or instance.

9.2 Overview of the MOSS Query System

The next lines describe the overall approach I adopted for MQS (MOSS Query System).

9.2.1 Input Query Format

Complex queries are defined as a tree in the graph of models defining the object base.

Examples:

```
Persons whose name is Dupond
("person" ("name" :is "Dupond"))

Persons whose name is Dupond, and who have a brother whose age is greater than 25
("person" ("name" :is "Dupond"))
  ("brother" ("person" ("age" > 25))))

Persons whose name is Dupond, and who have a brother whose age is greater than 25,
and who is not employed in a company not named UTC...
("person" ("name" :is "DUPOND"))
  ("brother" ("person" ("age" :between 25 30)))
  ("employer" ("company" ("name" :is-not "UTC")...)
```

Queries can be arbitrarily complex, provided they retain the form of a tree.

9.2.2 Basic Query Mechanism

The query system takes a formal query as input, focusing on a particular concept. It then computes a set of possible candidates, and tries to validate each candidate in turn by navigating *‡ la Prolog*.

9.2.3 Semantics of the Recorded Values

For the time being, MOSS does not record null values. A null value associated with a given property (attribute or relationship) means that MOSS does not know the value associated with the property (most of the time the property is not even recorded in the object). Thus there is no way of specifying currently that a property is known and that there is no value associated with it. For example, some persons from India have a family name but no first name. Thus we should store the attribute "first name" with a null value, meaning that there is no first name. This is currently not the case.

Thus, the answer to a query means that the objects are those known to MOSS to match the query constraints.

9.2.4 Implementation

The approach is implemented in three steps:

1. the formal query is parsed and replaced with a query using internal symbols, the syntax being checked for correctness;
2. a candidate list is computed using entry points in the query;
3. objects of the candidate list are checked in turn against the full query for validation. The result is a list of object identifiers.

A *query handler* object is defined to implement the query mechanism. Two MOSS methods?associated with the query handler?are available, depending on whether the user wants an interactive control, or a programmed?automatic?execution.

9.3 MOSS Query Syntax

The general query format is the following:

```
<general-query> ::= <entry-point> | <query>
<query>         ::= (<class> {<clause>}*)
<clause>        ::= (<simple-clause>) | (<or-clause>)
                     | (<or-constrained-sub-query>)
<simple-clause> ::= <tp-clause> | (<sub-query>)
<tp-clause>     ::= (<terminal-property> <tp-operator> <value>)
                     | (<terminal-property> <tp-operator> <variable>)
<or-clause>     ::= (OR <simple-clause>+)
<mixed-clause>  ::= (<sub-query>) | (<simple-clause> . {<mixed-clause>})
                                  | (<sub-query> . <mixed-clause>)
<sub-query>     ::= (<relationship> <cardinality-condition>  <query>)
<relationship>  ::= <structural-property> | <inverse-property>
<or-constrained-sub-query> ::= (OR <cardinality-condition> {<sub-query>}+)
<cardinality-condition> ::= (<numeric-operator> <arg>+)
<numeric operator> ::= < | <= | = | >= | > | <> | BETWEEN | OUTSIDE
<tp-operator>    ::= < | <= | = | >= | > | ? | BETWEEN | OUTSIDE
                         | IS | IS-NOT | IN | ALL-IN
                           | CARD= | CARD< | CARD<= | CARD> | CARD>=
<variable>       ::= symbol starting with ?
<entry-point>    ::= symbol | string

where * means 0 or more, and + one or more
```

9.3.1 Entry-Point Queries

An entry-point query consists of a symbol or a string. For example:

```
Asking for any object that has an entry point of DUPOND
DUPOND or "Dupond"
```

As an answer to this question, MOSS will simply collect all objects that have DUPOND as an entry-point (obtained from applying the default make-entry-point to the symbol or the string) of any possible attribute with entry-points. It will return a list of such objects.

9.3.2 Class Queries

This sort of query is used to obtain all elements from a given class. E.g., all persons:

```
(person) or ("person")
```

MOSS will return all elements of the class PERSON (individuals corresponding to the concept PERSON, including all elements of the subclasses of persons, e.g., teachers, students, etc.

One can disable getting information from subclasses by setting a global variable (see advanced features).

9.3.3 Simple Queries

The following queries are the simplest possible queries, e.g.,

- All males

```
("person" ("sex" :is "m"))
```

- All persons named "BarthËs" and known not to be males

```
("person" ("name" :is "BarthËs") ("sex" :is-not "m"))
```

Notice that putting several clauses amounts to having an implicit AND on such clauses.

- All persons with brothers

```
("person" ("brother" ("person")))
```

For such questions which do not include a cardinality constraint following the relationship, the parser adds a default condition (> 0), so that the query becomes:

```
("person" ("brother" (> 0) ("person")))
```

When MOSS processes the query, it starts with a set of possible candidates and examines each candidate in turn. For each candidate, it follows the has-brother relationship and is happy as soon as it finds a reference to a brother.

9.3.4 Queries with a Longer Path

This is the same as above but the query contains a longer path.

- persons who are cousins of a person who has a father with name "Labrousse"

```
(person (is-cousin-of
           (person (has-father
              (person (has-name :is "Labrousse"))))))
```

Note that we used the names of the concept and of the properties directly. When using strings an inverse property is written as "¿cousin". Hence we could write:

```
("person" (">cousin"
           ("person" ("father"
              ("person" ("name" :is "Labrousse"))))))
```

- persons who have a brother who is cousin of a person who has a father with name "Labrousse"

```
(person (has-brother
       (person (is-cousin-of
            (person (has-father
                  (person (has-name :is "Labrousse"))))))))
```

9.3.5 Queries with Disjunctions

- All persons who have at least a brother, or a sister

```
(person
     (OR  (has-brother (person))
          (has-sister (person))))
```

- All persons who have at least a daughter or a son, and a brother

```
(person
     (OR  (has-son (person))
          (has-daughter (person)))
     (has-brother (person)))
```

Note that all queries are set in conjunctive normal form. I.e., they appear as a conjunction of clauses, each clause being simple or a possible disjunction.

9.3.6 Queries with Cardinality Constraints (Including Negation)

The first example is a way to express negation:

- All persons with no (gettable or recorded) brothers (close world)

```
(person (has-brother (= 0) (person)))
```

The next examples include some sort of restrictions:

- All persons who have only one brother called "Sebastien"

```
(person (has-brother (= 1)
          (person (has-first-name is "Sebastien"))))
```

- All persons, brother of a person called "Sebastien" (inverse relationship)

```
(person (is-brother-of (= 1)
          (person (has-first-name is "Sebastien"))))
```

- All persons who have a father with more than 2 boys

```
(person (has-father (person (has-son (>= 2) (person)))))
```

9.4 Query Structure

A query is formulated as a tree, the nodes of which correspond to sets of objects?classes. A query can be represented as a *constrained graph*. Constraints are expressed as filters attached to the nodes. Except for queries that consist of a single entry point, all queries start with the name of a class. For example:

```
(PERSON (HAS-NAME :is "Dupond")
        (HAS-FIRST-NAME :is "Jean")
        (HAS-SISTER (PERSON (HAS-AGE < "20"))
        (HAS-EMPLOYER (COMPANY (HAS-ABBREVIATION :is "ABC"))))
```

The corresponding constrained graph has 3 nodes: the first one is the goal node corresponding to the objects answering the query; the second is a node for the persons that can be sisters of the persons we are looking for; the third one is a node for companies (Fig.9.1).

The goal is to find all persons named "Jean Dupond" with a sister less than 20, and who work for a company named ABC. To do so, one could select several strategies:

- S1: look at all persons in turn, select those who are named Dupond, and whose first name is Jean, then for all such persons check that they have a sister who is less than 20, then look for those who work in a company named ABC.

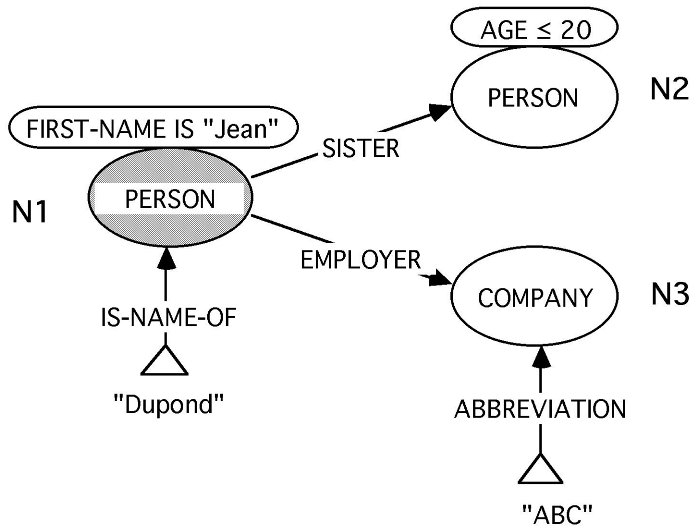

Figure 9.1: Node structure showing the goal node N1 and the related nodes: N2, N3. Constraints are shown in ovals, entry points are shown using triangles.

- S2: look at all persons who are less than 20, then look at all the persons who have such persons as sisters, then in the last lot check for those who work for ABC.
- S3: look at all employees of ABC, then keep those who have a sister who is less than 20.

Clearly, if our environment contains several thousands of persons, then strategy S1 is not very efficient, and it would be better to use strategy S2 or S3. In fact, here, ”Dupond” and ”ABC” are entry points and can be used to select only persons whose name is ”Dupond” or companies called ”ABC”.

Access Mode: In practice MOSS uses the entry-points to first select a subset of the class of persons, those that are named ”Dupond” and who work for the company ”ABC”. Then, for each person in this starting subset, the full query is tested, i.e. the first-name must be ”Jean”, and the person must have a sister with age less than 20. Only objects satisfying all conditions are put into a candidate list.

Each object of the candidate list is checked progressively, using the various links. On this example we would check first that a given person Pi in the candidate set has ”Jean” as a first name, then has a sister with age less than or equal to 20 (we stop checking as soon as we have found a sister satisfying the constraint, or if there is no such sister, in which case the query fails for Pi and Pi is discarded), then is working for a company names ”ABC”.

9.5 Access Algorithm

The purpose of the algorithm is to minimize the number of accesses to objects that must be visited.

At a given node access to objects can be done locally, either by means of entry-points if there are any, or by enumerating the objects in the class and subclasses (default behavior) represented by the node, or via the neighboring nodes when the query contains subqueries. However, in the latter case, access by means of inverse properties can only be done when the cardinality constraint is positive, i.e. (> 0) or $(>= 1)$, meaning that there should be at least one neighbor (default). When the cardinality constraint is different, e.g. $(= 0)$ or $(= 2)$ or (> 3) or $(BETWEEN 25)$, then such constraints must be used a posteriori and access must be done in the first node. The case of OR clauses is more complex.

9.5.1 Algorithm

When the query reduces to a single entry point then the answer is obtained immediately. Otherwise, we use the following algorithm.

Step 1 - Estimate a list of candidates

Use the query entry points in positive branches to compute a subset of the class of the goal node. The result may be *none* meaning no answer, a list of candidates, or nil, meaning that we must do a class access (i.e. load all the objects of the class).

Step 2 - Collect objects

For each object in the candidate list obtained in Step 1, navigate through the constrained graph and apply all filters to determine if the object must be kept in the final solution.

Thus, we start with a list of candidates, and a global test on the subset of objects to be checked. Possible test operators are ALL (for checking all objects), $<, <=, =>, >=, <>$, BETWEEN, OUTSIDE. If the test is verified, then we return a list of the accepted candidates; otherwise we return nil.

For each node of the constrained graph, corresponding to a particular set of objects, we operate as follows:

- First, we split the query into simple clauses?from which we dynamically build and compile a local filter?, and OR-clauses and sub-queries.
- At each cycle, we have a list of candidates waiting to be examined, and a good-list of objects which were candidates and which passed the tests.

  applying the global test to the good-list.

  If we have an immediate success, then we return the set of objects in the good-list.

  If we have a failure, we return nil.

  If we cannot decide, then we examine the current candidate.

  examining the current candidate:
  we first apply the local filter. If the test fails, then we proceed with the next candidate in the candidate list.

  Then, we must check the set of all clauses (conjunction of OR-clauses and sub-queries):

  if there are no more clauses to check, we have a success and we proceed with the next candidate

  if the clause is a sub-query, we proceed recursively.

  if the clause is an OR-clause, we check each sub-clause in turn until we find one that is verified.

9.6 Constraints on the Relationship Cardinality

Assume now that we want to put some constraint on the cardinality of a relationship, e.g. we want to know the persons named Jean Dupond, who work for ABC, and who have exactly 2 sisters with age less than 20. This can be expressed as:

```
(PERSON (HAS-NAME :is "Dupond")
        (HAS-FIRST-NAME :is"Jean")
        (HAS-SISTER (= 2)(PERSON (HAS-AGE < "20"))
        (HAS-EMPLOYER (COMPANY (HAS-ABBREVIATION :is "ABC"))))
```

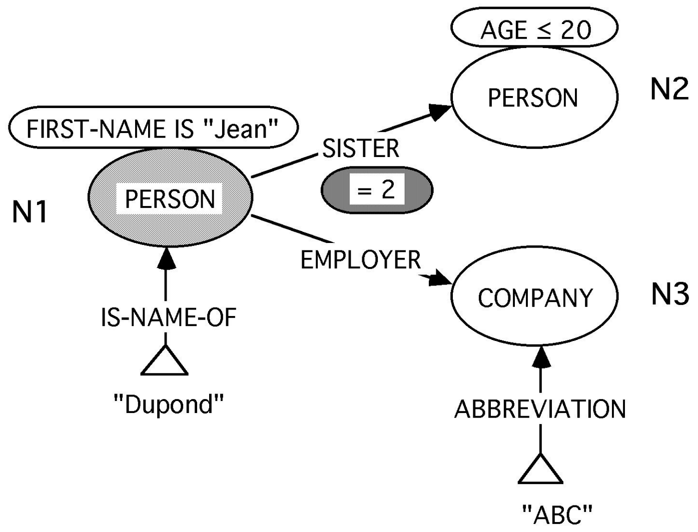

Figure 9.2: Constrained graph showing an intermediate cardinality filter

The constrained graph in this case is very similar to that of Fig.9.1 but for the fact that we have now a constraint on the cardinality of the sister relationship.

In this case, we collect sisters as previously and we can distinguish two cases: (i) we find more than 2 sisters with age less than or equal to 20, in which case the corresponding starting object fails to meet the query requirements and is thus removed from the solution list; (ii) in the other case we must wait until we have examined all sisters to check for their number.

9.7 Queries Containing OR Clauses

OR clauses are defined in a restricted way in MOSS. Indeed, they can involve simple clauses (mix of tp-clause and sub-query) or constrained sub-queries. This section intends to give several examples and relate them to access considerations.

9.7.1 Simple tp-clauses without entry points

Example

```
(OR (HAS-HAIR-COLOR :IS "Red") (HAS-AGE > 22))
```

It is not possible to use this clause to infer a list of possible objects. The two clauses will be used to filter a list of possible candidates.

9.7.2 Simple tp-clauses with entry points

Example

```
(OR (HAS-NAME :IS "Dupond") (HAS-COUNTRY :IS "France"))
```

Assuming that NAME and COUNTRY are entry points for a person, means that we can obtain the list of objects whose name is DUPOND and country is FRANCE. Merging the two lists yields a list of possible candidates. Note that we can do this only because each clause has an ?IS? operator.

9.7.3 Simple tp-clause with and without entry points

Example

```
(OR (HAS-NAME :IS "Dupond") (HAS-HAIR-COLOR :IS "Red"))
```

DUPOND could yield a list of possible candidates. However, HAIR-COLOR brings no specific restriction on the set of possible objects of the class. Thus we cannot retrieve any restricted list of possible candidates from this clause.

9.7.4 Sub-queries

Example

```
(OR (HAS-BROTHER (PERSON (HAS-NAME :IS "Dupond")))
    (HAS-EMPLOYER (COMPANY (HAS-NAME :IS "ABC")))))
```

In this case, each branch of the OR clause can yield objects. By accessing persons who are brothers of persons named DUPOND and persons who work for the ABC company, we can fuse those two lists that will provide a final list of possible candidates.

9.7.5 Constrained sub-queries

Example

```
(OR (> 2) (HAS-BROTHER (PERSON))
          (HAS-SISTER (PERSON)))
```

meaning than we can accept any combination of brothers and sisters provided there are more than 3.

Example

```
(OR (> 2) (HAS-BROTHER (= 2) (PERSON))
          (HAS-SISTER (PERSON)))
```

in that case the number of brothers (if any) must be exactly 2.

Only sub-queries can be constrained this way. If we mix sub-queries and tp-clauses in an or-constrained clause the semantics becomes tricky. This MOSS does not allow such mixed constrained OR-clauses.

If the cardinality constraint is positive, i.e. requires at least one value, then we can draw the same conclusion as in 9.7.4. Indeed, if the clause must be validated, it will necessarily involve some of the objects found in the branches of the sub-queries. Hence, such objects can be used to recover a list of possible candidates by following the inverse-property link.

9.7.6 Default sub-query constraint

We have shown sub-queries without cardinality conditions, e.g.

```
(OR (HAS-BROTHER (PERSON (HAS-NAME :IS "Dupond")))
    (HAS-EMPLOYER (COMPANY (HAS-NAME :IS "ABC"))))
```

In practice the system adds automatically a default constraint (¿ 0) when there are none, which yields:

```
(OR (HAS-BROTHER (> 0) (PERSON (HAS-NAME :IS "Dupond")))
    (HAS-EMPLOYER (> 0) (COMPANY (HAS-NAME :IS "ABC"))))
```

9.7.7 Negative OR clauses

Negation is expressed by the (= 0) cardinality constraint.

Example

```
(OR (= 0) (HAS-BROTHER (= 2) (PERSON))
          (HAS-SISTER (PERSON)))
```

meaning that we do not want people who have 2 brothers or sisters. Now if we look at the query at a higher level we might have something like:

```
(PERSON (HAS-NAME :IS "Labrousse")
              (OR (= 0) (HAS-BROTHER (= 2) (PERSON))
                                (HAS-SISTER (> 0) (PERSON))))
```

which is actually equivalent to

```
(PERSON (HAS-NAME :IS "Labrousse")
                  (HAS-BROTHER (<> 2) (PERSON))
                  (HAS-SISTER (= 0) (PERSON))))
```

Therefore, in this case the OR clause can be simplified provided we switch the attribute operators to their complement. This operation is done as a preprocessing step of the formal query.

9.7.8 Conclusion

When dealing with OR clauses, some conclusions regarding the possible object candidates can be drawn in some specific cases as explained in the previous paragraphs. This is taken into account by the find-subset function. However, most of the time, the clauses and sub-queries will be used a posteriori on a list of candidates generated differently.

9.8 Multiple Values

Having multiple values in the attributes and relationships introduces some difficulty.

Semantics of Multiple Values

The semantics of a multiple value was not made explicit. One must be careful. A multiple value associated with a structural property or relationship means that the link exists between all objects involved in the relationship; e.g.

```
(...(HAS-COUSIN <person-32> <person-48>)...)
```

means that the person has 2 cousins.

For an attribute, this might be different when one is not careful, e.g.

```
(...(HAS-FIRST-NAME "John" "Julius")
```

has the same meaning as above, but a polyline with 3 vertices expressed as:

```
(...(HAS-VERTEX (0 0) (25 100)(0 20))...)
```

may have a different semantics if it is assumed that the middle vertex is an intermediate point in the polyline, and the other ones are end-points. Here, we no longer have a set, but a structured value. Hence, it should be recorded as a single value being a list of ver-tices and not as a multiple value.

```
(...(HAS-VERTEX ((0 0) (25 100)(0 20)))...)
```

Using a set semantics, we still have problems with negative operators on multiple values e.g.

```
(HAS-NAME IS-NOT "Albert")
```

on a value like

```
("Albert" "Joseph")
```

since the generic operator that builds filters is designed for positive operators, thus makes an implicit OR, while with an operator like not-equal we mean an implicit AND. The situation is even more confusing with range operators like $3 < xx < 45$.

To solve the problem we must specify the semantics of multiple values (which depends on the nature of the associated property).

Multiple values mean alternative values (equivalent values).

Hence when we have a positive operator like :is, it means equal to one of any of the values, when it is :is-not (not-equal), it means equal to none of the values. In case of multiple values we could have more specific operators like NOT-ONE, NOT-ANY, <ALL, <ONE, etc.

The idea is to specify the range of operators to build the overall filtering function

9.9 Querying the Database Model

Querying the database model is not different from querying the data. E.g., if we want to obtain all classes which have a =summary method, we can use the following query:

```
(entity (has-method
          (method (has-method-name is "=summary"))))
```

9.10 Programmer's Interface

The programmer's interface uses several classes and methods described in this section. In practice however, one uses the access function.

9.10.1 Classes

QUERY-HANDLER

The QUERY-HANDLER class was designed to provide the user with a few methods for querying the object base. It has no properties. MOSS uses an instance of query-handler, bound to the global variable `*query-handler*` or `*qh*`.

QUERY

The QUERY class is intended to save queries. Hence queries are reified, and can be referenced at a later stage.

A query has the following attributes:

QUERY-NAME: single value that will be used as an entry point

USER-EXPRESSION: the raw query as given by the user

EXPRESSION: the parsed query, containing internal references.

ANSWER: the answer to the query if the user wants to save it.

9.10.2 Methods

9.11 Advanced features

9.11.1 Disabling Access to Sub-Classes

It is possible to disable access to subclasses, by setting the global variable `*query-allow-sub-classes*` to nil.

9.11.2 Query with Co-Reference Variables

Such queries have Prolog-like variables:

- All persons who have a cousin of the same sex

```
(person (has-sex :is ?x)
        (has-cousin (person (has-sex :is ?x))))
```

When encountered for the first time, the variable must be on the right side of an equality operator. Other references are unrestricted, e.g.

```
(person (has-sex :is ?x)
        (has-cousin (person (has-sex :is-not ?x))))
```

- All persons who have a son with the same name as the son of his brother

```
(person (has-son  (person (has-first-name is ?X)))
        (has-brother
          (person (has-son
                    (person (has-first-name is ?X))))))
```

- All persons with age greater than 22, having a younger sister

```
(person (has-age > 22)
        (has-age is ?x)
        (has-sister (person (has-age <= ?x))))
```

Danger

The meaning of such queries appear to be intuitively simple to grasp. However, this is not the case as demonstrated by the following example:

```
(person (has-son (person (has-age is ?x))
         (has-daughter (person (has-age <= ?x))))
```

Does this query means in the case when a person has several sons and one daughter that the daughter must be younger than one of the sons, or younger than each son? What about the case when we have several sons and several daughter?

For the time being, we decide, that in a Prolog style, the query will succeed when a person has at least one daughter younger than one of his/her sons

9.11.3 Class Variables

Class variables are not yet implemented.

9.12 Appendix A - Brief Comparison with SQL

This paragraph is not intended to give a detailed comparison between MQS and SQL used for the relational databases, but to illustrate a few points.

9.12.1 Major Difference between MQS and SQL

Objects vs. Relations

The major difference is that MQS applies to objects and SQL applies to relational tables (i.e., values of attributes).

MQS selects objects in a database and always returns a set of objects, SQL constructs a new relational table by combining pieces of other tables, according to the relational algebra.

Access vs. access and updates

MQS is used to access objects, SQL can be used to create, to access, or to modify objects.

Thus, in order to be able to compare MQS with SQL, we only take from SQL the accessing part of the language, stripping the modifying and the formatting possibilities. This, in effect reduces the language to a few basic constructs, like:

```
SELECT FROM from-item-list
       WHERE predicate
```

9.12.2 Example

Consider the following relational tables taken from (Date 87) [1].

Suppliers: s

sno	sname	status	city
s1	Smith	20	London
s2	Jones	10	Paris
s3	Blake	30	Paris
s4	Clark	20	London

Part number: p

pno	pname	color	weight	city
p1	Nut	Red	12	London
p2	Bolt	Green	17	Paris
p3	Bolt	Blue	17	Rome
p4	Scew	Red	14	London

Project number: j

jno	jname	city
j1	Sorter	Paris
j2	Punch	Rome
j3	Reader	Athens
j4	Console	Athens
j5	Collator	London

Shipman: spl

[1] C.J. Date, A Guide to INGRES, Addison Wesley, 1987

sno	pno	jno	qty
S1	P1	J1	200
S2	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S3	P3	J1	300
S3	P4	J2	500
S4	P6	J3	300

The relational tables in fact could represent MOSS objects, where the numbers could be considered to correspond to the MOSS internal identifiers of the objects.

Get supplier names from suppliers who supply at least one red part

SQL

```
SELECT DISTINCT S.NAME
FROM S, SP, P
WHERE S.SNO = SP.SNO
AND SP.PNO = P.PNO
AND P.COLOR = 'RED'

SELECT S.NAME
FROM S
WHERE S.SNO IN
( SELECT SP.NO
  FROM SP
  WHERE SP.NO IN
( SELECT P.PNO
         FROM P
         WHERE P.COLOR = 'RED' )
```

MQS

```
(supplier (has-part (part (has-color is "red"))))
```

In the case of MQS, the names must be further retrieved from the resulting list of suppliers.

Get supplier names for suppliers who supply part P2

SQL

```
SELECT S.NAME
FROM S
WHERE S.SNO IN
( SELECT SP.NO
  FROM SP
  WHERE SP.NO = 'P2' )
```

MQS

```
(supplier (has-part (part (has-number is "P2"))))
```

Suppliers who supply at least 200 parts

SQL

```
SELECT S.NAME
FROM S, SP
WHERE S.SNO = SP.SNO
AND SP.QTY >= 200
```

MQS

```
(supplier (has-shipment (shipment (has-quantity >= 200))))
```

Suppliers who supply at least 200 red parts

SQL

```
SELECT S.NAME
FROM S, SP, P
WHERE S.SNO = SP.SNO
AND S.SNO = P.PNO
AND SP.QTY >= 200
AND P.COLOR = 'RED'
```

MQS

```
(supplier
    (has-shipment
        (shipment (has-quantity >= 200)
                  (has-part (part (has-color is "red")))))
```

9.13 Appendix B - Implementation

This section gives information about how the query mechanism is implemented.

9.13.1 Overall View of the Query Process

The query is given to the access function that implements the following algorithm:

1. If the query is a symbol or a string, then it is an entry point and it is processed directly by the access-from-entry-point function.
2. Then, if not already parsed, the query is parsed and transformed into an internal format. If errors are found the query returns nil.
3. A set of candidates is computed from the parsed query by calling the find-subset function. The function uses entry points to collect a set of possible candidates, or if there are no useful entry points in any branch of the query synthesizes the list of instances of the target class.
4. If there are no candidates, return nil.
5. Otherwise, call the function filter-candidates that determines whether a list of candidates (instance of the target class) satisfies the constraints imposed by the query.

6. Remove the dead objects
7. Return the list of remaining objects.

The following sections detail the various steps.

9.13.2 Entry-Point Queries

Entry points are computed using the make-all-possible-entry-points function that collects all =make-entry methods and apply it to the entry point.

```
? (moss::make-all-possible-entry-points "barthes")
(HAS-BARTHES BARTHES)
```

Here the string "barthes" is interpreted as the possible name of a property of the name of a person. Of course "barthes" cannot be the name of a property. However, the situation is different in the following case:

```
? (moss::make-all-possible-entry-points " person name")
(HAS-PERSON-NAME PERSON-NAME)
```

Then all objects corresponding to the computed entry points are collected by the %%get-objects-from-entry-point function and the list of internal identifiers of the objects is returned.

Examples

```
? (MOSS::ACCESS-FROM-ENTRY-POINT "barthËs")
($E-PERSON.1 $E-PERSON.2 $E-PERSON.3 $E-STUDENT.1 $E-PERSON.4 $E-PERSON.5
             $E-PERSON.6 $E-PERSON.7 $E-PERSON.8 $E-PERSON.9 $E-PERSON.10 $E-PERSON.11)

? (MOSS::ACCESS-FROM-ENTRY-POINT "name")
($T-NAME $T-PERSON-NAME $T-ORGANISM-NAME)
```

9.13.3 Parsing a Query

Parsing a query consists in checking the syntax of the query and transforming it into a query with internal format. This is done by the parse-user-query function.

Examples

```
? (moss::parse-user-query
   '(person (has-name is "barthes")
     (has-brother (person (has-first-name is "antoine")))))

($E-PERSON ($T-PERSON-NAME :IS "barthes")
 ($S-PERSON-BROTHER (> 0) ($E-PERSON ($T-PERSON-FIRST-NAME :IS "antoine"))))

? (moss::parse-user-query
   '(person (">cousin" (person (has-first-name is "antoine")))))

($E-PERSON ($S-PERSON-COUSIN.OF (> 0)
               ($E-PERSON ($T-PERSON-FIRST-NAME :IS "antoine"))))
```

Queries including negations are transformed into queries containing negative operators. In addition, parsing the query includes expanding the virtual classes references.

9.13.4 Collecting a First Subset of Candidates

This step is performed by the find-subset recursive function. The function returns a list of objects computed from the entry-points eventually found in the sub-query corresponding to its argument. When the function returns nil, then this means that there is no limit coming from the current branch (the function argument), when the function returns the symbol `*none*`, then this means that the current branch has no possible solution.

The function takes into account negative conditions as well as disjunctions.

Example

```
(moss::find-subset '($E-person
                     ($T-PERSON-NAME :is "barthes")
                     (or ($S-PERSON-BROTHER
                          (> 0) ($E-person ($T-person-name :is "barthes")))
                         ($S-PERSON-SISTER
                          (> 0) ($E-person ($T-person-name :is "labrousse"))))))
; returns a list of 7 objects
```

The role of this step is to find a minimal set of candidates to be checked in the next step by using the entry points (index objects) found in the various branches of the query if any.

9.13.5 Filtering Candidates

This is the crucial test implemented by the filter-candidates function.

The function takes a list of objects, a cardinality test to be applied to the objects once they are filtered and a path from the node representing the class of the objects, and bindings of coreferenced variables. The role of the function is to filter the objects, one at a time, checking that they obey the constraints represented by the path: e.g.

```
($E-PERSON ($S-BROTHER (> 1) ($E-PERSON)))
```

which means that we discard objects that have not at least 2 brothers.

As objects are checked they are added to a good-list. Each time we add an object to the good-list we can check the cardinality conditions for failure (e.g. we want less than 2 cousind and have already found a second one, in which case it is not necessary to access more cousins. However, sometimes we want to access all objects that conform to the path. In that case we simply disallow the early success test and keep filtering all the objects.

Now the question is: what do we return?

Sometimes we want the list of objects (e.g. at toplevel), at other times we simply want to know if the candidates pass the cardinality test, in which case a simple T/nil answer is acceptable. Note that we cannot use the list of objects as an answer to this purpose, since the list may be empty, but the test (= 0) could be satisfied. The idea is to return 2 values: a T/NIL value and the list of objects.

Bindings When we call the function, we may have already a set of instantiated coreference variables. They will be used to check the values. E.g.

```
($E-PERSON ($S-COUSIN (> 1) ($E-PERSON ($T-HAS-NAME :is ?x))))
```

with bindings

```
((?x "Labrousse")(?y "BarthËs"))
```

If the path contains simple clauses with variables, then the value of bindings will be changed temporarily for checking the rest of the path. However, on return, the value of bindings is not modified and thus need not be returned. Indeed, we'll check the next object with the same initial bindings as as the previous one.

Node Filters

At each node of the query tree, we must collect objects and apply filters corresponding to the clauses of the query. The function split-query-and-compute-filters prepares the stage. The function splits the query into different parts, and compute a filter from attribute clauses and simple OR attribute clauses to apply to future candidates. If the compiler is available, then compiles the filter.

When a cardinality test is provided, builds another filter to test that will apply to the number of retrieved objects. Those operations should be done only once and not everytime a new candidate is examined, which is the justification for this function.

The arguments to the function are:

- query: sub-query, e.g.

```
($E-PERSON ($T-PERSON-NAME :IS \"Barthes\")
        ($S-PERSON-COUSIN (> 0)
          ($E-PERSON ($T-PERSON-FIRST-NAME :IS \"antoine\"))))
```

- cardinality-filter: something like (> 2) or (:BETWEEN 4 5) or nil

The function returns 6 values:

- name of local filter
- name of cardinality filter (can be nil)
- clauses-with-variables
- or-clauses
- subqueries
- class variable

Examples

```
? (moss::split-query-and-compute-filters
   '($E-PERSON ($S-PERSON-COUSIN (> 0)
                              ($E-PERSON ($T-PERSON-NAME :IS "Labrousse"))))
   )
(#:LOCAL-FILTER-14023
 #:TEST-14024
 NIL
 (($S-PERSON-COUSIN (> 0) ($E-PERSON ($T-PERSON-NAME :IS "Labrousse"))))
 NIL)

  ;;; when compiler is not available:
? (moss::split-query-and-compute-filters
   '($E-PERSON ($S-PERSON-COUSIN (> 0)
                              ($E-PERSON ($T-PERSON-NAME :IS "Labrousse")))))
```

```
((LAMBDA (MOSS::XX)
   (AND (INTERSECTION '($E-PERSON $E-STUDENT) (MOSS::%TYPE-OF MOSS::XX)) MOSS::XX))
 (LAMBDA (MOSS::XX) (IF (> (LENGTH MOSS::XX) 0) MOSS::XX))
 NIL
 (($S-PERSON-COUSIN (> 0) ($E-PERSON ($T-PERSON-NAME :IS "Labrousse"))))
 NIL)
```

Then filters are applied to the candidates at each node recursively.

We have different types of filters.

Clause Filters

Clause filters apply to an attribute clause, e.g.

```
? (make-clause-filter '$T-YEAR '<= 2002)
((LAMBDA (ZZ)
   (AND ZZ
        (EVERY #'(LAMBDA (YY)
                   (<= (convert-data-to-number YY) '2002))
               ZZ)))
 (=> XX '=GET-ID '$T-YEAR))

? (make-clause-filter '$T-YEAR '= 2002 'some)
((LAMBDA (ZZ)
   (AND ZZ
        (SOME #'(LAMBDA (YY)
                  (= (convert-data-to-number YY) '2002))
              ZZ)))
 (=> XX '=GET-ID '$T-YEAR))
```

Filters for Multiple Values

When dealing with multiple values we must produce the ad hoc filters, e.g.

Examples

```
? (moss::make-local-filter-for-multiple-values
   '("barthËs" "biesel")
   '$T-PERSON-NAME
   :is)
((LAMBDA (ZZ)
   (AND ZZ
        (INTERSECTION '("BARTHES" "BIESEL")
                      (MAPCAR #'NORMALIZE-VALUE ZZ)
                      :TEST
                      #'EQUAL)))
 (=> XX '=GET-ID '$T-PERSON-NAME))

? (moss::make-local-filter-for-multiple-values
   '("barthËs" "biesel")
   '$T-PERSON-NAME
```

```
   :is-not)
((LAMBDA (ZZ)
   (AND ZZ
        (NOT (INTERSECTION '("BARTHES" "BIESEL")
                           (MAPCAR #'NORMALIZE-VALUE ZZ)
                           :TEST
                           #'EQUAL))))
 (=> XX '=GET-ID '$T-PERSON-NAME))

? (moss::make-local-filter-for-multiple-values
   '(22)
   '$T-PERSON-AGE
   :equal)
((LAMBDA (ZZ)
   (AND ZZ (SOME #'(LAMBDA (YY) (= (convert-data-to-number YY) '22)) ZZ)))
 (=> XX '=GET-ID '$T-PERSON-AGE))

? (moss::make-local-filter-for-multiple-values
   '(("barthËs" "biesel"))
   '$T-PERSON-NAME
   :all-in)
((LAMBDA (ZZ)
   (AND ZZ
        (EVERY #'(LAMBDA (YY) (MEMBER YY ZZ :TEST #'EQUAL)) '("BARTHES" "BIESEL"))))
 (NORMALIZE-VALUE (=> XX '=GET-ID '$T-PERSON-NAME)))
```

Multiple Filters

Multiple filters act on conjunctions or disjunctions of properties, e.g.

Examples

```
? (moss::make-multiple-filter
     '(($T-PERSON-NAME :is-not "Labrousse")
       ($T-AGE <= 23)))
(((LAMBDA (MOSS::ZZ)
    (AND MOSS::ZZ
         (NOT (INTERSECTION '("LABROUSSE")
                            (MAPCAR #'MOSS::NORMALIZE-VALUE MOSS::ZZ)
                            :TEST
                            #'EQUAL))))
  (MOSS::=> MOSS::XX '=GET-ID '$T-PERSON-NAME))
 ((LAMBDA (MOSS::ZZ)
    (AND MOSS::ZZ
         (NOT (INTERSECTION '(23)
                            (MAPCAR #'MOSS::NORMALIZE-VALUE MOSS::ZZ)
                            :TEST
                            #'<))))
  (MOSS::=> MOSS::XX '=GET-ID '$T-AGE)))

? (moss::make-multiple-filter
```

```
     '((OR ($T-PERSON-NAME :is-not "Labrousse")
       ($T-AGE <= 23))))
((OR ((LAMBDA (MOSS::ZZ)
        (AND MOSS::ZZ
             (NOT (INTERSECTION '("LABROUSSE")
                                (MAPCAR #'MOSS::NORMALIZE-VALUE MOSS::ZZ)
                                :TEST
                                #'EQUAL))))
      (MOSS::=> MOSS::XX '=GET-ID '$T-PERSON-NAME))
     ((LAMBDA (MOSS::ZZ)
        (AND MOSS::ZZ
             (NOT (INTERSECTION '(23)
                                (MAPCAR #'MOSS::NORMALIZE-VALUE MOSS::ZZ)
                                :TEST
                                #'<))))
      (MOSS::=> MOSS::XX '=GET-ID '$T-AGE))))
```

Then, the filters are applied to the candidate, and if the candidate passes it is inserted into a good-list.

9.13.6 Examples of Queries

This section gives some examples of queries that one can test against the family file.

```
;;; test on family data
;;; macro to print the results
(defMacro pp ()
  `(progn (mapcar #'(lambda (xx) (print (send xx '=summary))) *)
          nil))

(moss::access '(person (has-brother (= 0) (person))))
;;;   access all males
(moss::access '(person (has-sex is "m")))
;;; ===== For the next ones use MOSS window...
;;;   access all person named "BarthËs" and not known to be males
(moss::access '(person (has-name is "BarthËs")(has-sex is-not "m")))
;;;   access all persons with brothers
(moss::access '(person (has-brother (person))))
;;;   access all persons with no (gettable or recorded) brothers (close world)
(moss::access '(person (has-brother (= 0) (person))))
;;;   access all Barthes's who have only one brother called "Sebastien"
(moss::access
 '(person (has-name is "barthes")
   (has-brother (= 1) (person (has-first-name is "Sebastien")))))
;;;   access all persons brother of a person called sebastien
(moss::access
 '(person (is-brother-of (= 1) (person (has-first-name is "Sebastien")))))
;;;   access all persons with at least 2 children of any sex
(moss::access
 '(person (or (>= 2) (has-son (>= 0) (person))
              (has-daughter (>= 0) (person)))))
```

```
;;; answer should be: jpb dbb pxb chb pb mb mbl ml
;;; the following question removes jpb and dbb from the list. Indeed, one must
;;; have at least 2 sons if any, or 2 girls.
(moss::access
 '(person (or (>= 2) (has-son (>= 2) (person))
              (has-daughter (>= 0) (person)))))
;;; answer should be: pxb, chb, pb, mb, mbl, ml
;;; The following question removes mlb and ml, since we want at most 2 girls.
(moss::access
 '(person (or (>= 2) (has-son (>= 2) (person))
              (has-daughter (<= 2) (person)))))
;;; answer should be: pxb, chb, pb, mb
;;; next query has a bad syntax (cannot include TP-clause in a constrained sub-query)
(moss::access
 '(person (or (>= 2) (has-son (>= 2) (person))
              (has-sex = "f")
              (has-daughter (<= 2) (person)))))
;;; correct syntax should be. However it is not accepted (recursive OR)
(moss::access
 '(person (or (has-sex is "f")
              (or (>= 2) (has-son (>= 2) (person))
                  (has-daughter (<= 2) (person))))))
;;; access all persons whith at least 2 children and most 2 girls and a boy
;;; answer is jpb, dbb
(moss::access
 '(person (or (>= 2) (has-son (<= 1) (person))
              (has-daughter (<= 2) (person)))))
;;;   access all persons with at least 3 children one of them being a son
;;; note that we cannot do that with a single OR clause.
;;; answer is pxb, chb, pb, mb
(moss::access
 '(person (or (>= 3) (has-son (>= 0) (person))
              (has-daughter (>= 0) (person)))
   (has-son (>= 1) (person))))
;;;   persons who are cousins of a person who has a father with name "Labrousse"
;;; answer is: psb cxb ab sb eb
(moss::access
 '(person (is-cousin-of
           (person (has-father
                    (person (has-name is "Labrousse")))))))
;;;   persons who have a brother who is cousin of a person who has a father...
;;; answer is: psb sb eb ab
(moss::access
 '(person (has-brother
           (person (is-cousin-of
                    (person (has-father
                             (person (has-name is "Labrousse")))))))))
;;;   persons who have a father with more than 1 boy
;;; answer is: jpb pxb ab sb eb mglb
(moss::access
```

```
 '(person (has-father (person (has-son (>= 2) (person))))))
;;;   persons who have  a father with  at least 3 children with at least one girl
;;;           and whose father has a brother
;;; answer: ab, sb, eb
(moss::access
 '(person (has-father
           (person   ; condition on the cardinality of the subset!
            (or (>= 3) (has-son (> -1) (person))
                (has-daughter (> -1) (person)))
            (has-daughter (>= 1) (person))
            (has-brother (person))))))
;;; persons who have 2 names: dbb, mgl
(moss::access
 '(person (has-name card= 2)))
(moss::access
 '(person (has-name is "labrousse")(has-name card= 2)))
;;; persons with more than 1 first name: jpb, cxb, chb (!), psb
(moss::access
 '(person (has-first-name card>= 2)))

;;; ------------------- Queries with coreference varaibles
;;;   person who has a cousin of the same sex
;;; answer should be: cxb ab sb sl cl al
(moss::access
 '(person (has-sex is ?x) (has-cousin (person (has-sex is ?x)))))
;;;   person who has a son with the same name as the son of his brother
;;; illegal query since one cannot set coreference variables in a wub-query
(moss::access
 '(person (has-son  (person (has-first-name is ?X)))
   (has-brother
    (person (has-son  (person (has-first-name is-not ?X)))))))
```

9.14 Appendix C - The Family Knowledge Base

The family knowledge base is the one used in this document. It is a fairly old test file that contains persons, organisms and links between such persons and organisms. Family links are complex enough to generate interesting queries.

The family knowledge base is produced in the common-graphics or common-lisp package. Four concepts are defined. Two macros, mp and ms, are defined to ease the input of individuals, a small expert system is defined to close the family links.

```
;;;===============================================================================
;;;06/08/33
;;; F A M I L Y (File FAMILY-Mac.LSP)
;;;
;;; Copyright Jean-Paul Barthès @ UTC 1994
;;;
;;;===============================================================================

;;; This file creates a family for checking the LOB window system, and the query
```

```
;;; system. Thus it requires the corresponding modules.
;;;1995
;;;  2/13 Adding relationships in the family to be able to use the file for
;;;       testing the query system
;;;       Adding other persons not actual members of the family
;;;  2/18 Adding a mechanism for saturating the family links -> v 1.1.1
;;;  2/19 Adding the age property to all persons and birth date (year)
;;;       Changing the version name to comply with the curreunt MOSS version 3.2
;;;       -> v 3.2.2
;;;  2/25 Adding orphans -> v 3.2.3
;;;1997
;;;  2/18 Adding in-package for compatibility with Allegro CL on SUNs
#|
2003
 0517 used to test changes to the new MOSS v4.2a system
2004
 0628 removing ids from the class definitions
 0715 adding :common-graphics-user-package for ACL environments
 0927 changinging the Ë and Ë characters to comply with ACL coding...
2005
 0306 defining a special file for mac because of the problem of coding the French
      letters
 1217 Changing the file to adapt to Version 6.
2006
 0822 adding count to the inference loop
|#

#+MCL
(in-package "COMMON-LISP-USER")
#+(AND MICROSOFT-32 IDE)
(in-package :common-graphics-user)
#-IDE
(in-package :common-lisp-user)

#+(or ALLEGRO MICROSOFT-32)
(eval-when (:load-toplevel :compile-toplevel)
  (use-package :moss))

(moss::mformat "~%;*** loading Family 3.4 Tests ***")

(eval-when (load compile eval)
  (use-package :moss))

;;; remove tracing while objects are built
(moss::toff)

(setq moss::*allow-forward-references* nil)

(defconcept (:en "PERSON" :fr "PERSONNE")
  (:doc "Model of a physical person")
```

```
  (:tp (:en "NAME" :fr "NOM") (:min 1)(:max 3)(:entry))
  (:tp (:en "FIRST-NAME" :fr "PRENOM"))
  (:tp (:en "NICK-NAME" :fr "SURNOM"))
  (:tp (:en "AGE" :fr "AGE") (:unique))
  (:tp (:en "BIRTH YEAR" :fr "ANNEE DE NAISSANCE") (:unique))
  (:tp (:en "SEX" :fr "SEXE") (:name sex)(:unique))
  (:sp (:en "BROTHER" :fr "FRERE") "PERSON")
  (:sp (:en "SISTER" :fr "SOEUR") "PERSON")
  (:rel (:en "HUSBAND" :fr "MARI") "PERSON")
  (:rel (:en "WIFE" :fr "FEMME") "PERSON")
  (:rel (:en "MOTHER" :fr "MERE") "PERSON")
  (:rel (:en "FATHER" :fr "PERE") "PERSON")
  (:rel (:en "SON" :fr "FILS") "PERSON")
  (:rel (:en "DAUGHTER" :fr "FILLE") "PERSON")
  (:rel (:en "NEPHEW" :fr "NEVEU") "PERSON")
  (:rel (:en "NIECE" :fr "NIECE") "PERSON")
  (:rel (:en "UNCLE" :fr "ONCLE") "PERSON")
  (:rel (:en "AUNT" :fr "TANTE") "PERSON")
  (:rel (:en "GRAND-FATHER" :fr "GRAND PERE") "PERSON")
  (:rel (:en "GRAND-MOTHER" :fr "GRAND MERE") "PERSON")
  (:rel (:en "GRAND-CHILD" :fr "PETITS-ENFANTS") "PERSON")
  (:rel (:en "COUSIN" :fr "COUSIN") "PERSON")
  )

(defconcept (:en "COURSE" :fr "COURS")
  (:tp (:en "TITLE" :fr "TITRE") (:unique))
  (:tp (:en "LABEL" :fr "CODE") (:entry))
  (:doc "Course followed usually by students")
  )

(defconcept (:en "STUDENT" :fr "ETUDIANT")
  (:is-a "PERSON")
  (:rel (:en "COURSES" :fr "COURS") "COURSE")
  )

(defconcept (:en "ORGANISM" :fr "ORGANISME")
  (:tp (:en "NAME" :fr "NOM"))
  (:tp (:en "ABBREVIATION" :fr "SIGLE") (:entry))
  (:rel (:en "EMPLOYEE" :fr "EMPLOYE") "PERSON")
  (:rel (:en "STUDENT" :fr "ETUDIANT") "STUDENT")
  )

;;;(defownmethod =make-entry (HAS-NAME HAS-PROPERTY-NAME TERMINAL-PROPERTY
;;;                                     :class-ref PERSON)
;;;  (data)
;;;  "Builds an entry point for person using standard make-entry primitive"
;;;  (moss::%make-entry-symbols data))

(defownmethod =if-needed (AGE HAS-PROPERTY-NAME ATTRIBUTE :class-ref PERSON)
  (obj-id)
```

```
  (let ((birth-year (car (send obj-id '=get 'HAS-BIRTH-YEAR))))
    (if (numberp birth-year)
      ;; do not forget, we must return a list of values
      (list (- (get-current-year) birth-year)))))

(definstmethod =summary PERSON ()
  "Extract first names and names from a person object"
  ;(append (HAS-FIRST-NAME) (HAS-NAME))
  (let ((result (format nil "~{~A~^ ~} ~{~A~^-~}" (Has-first-name) (has-name))))
    (if (string-equal result " ")
      (setq result "<no data available>"))
    (list result)))

;;; First define a macro allowing to create persons easily

(defMacro mp (name first-name sex var &rest properties)
  `(progn
     (defVar ,var)
     (defindividual "PERSON"
                    ("NAME" ,@name)
                    ("FIRST-NAME" ,@first-name)
                    ("SEX" ,sex)
                    (:var ,var)
                    ,@properties)))

(defMacro ms (name first-name sex var &rest properties)
  `(progn
     (defVar ,var)
     (defindividual STUDENT
                    ("NAME" ,@name)
                    ("FIRST-NAME" ,@first-name)
                    ("SEX" ,sex)
                    (:var ,var)
                    ,@properties)))

;;; Instances

(mp ("BarthÀs") ("Jean-Paul" "A") "m" _jpb ("BIRTH YEAR" 1945))
(mp ("BarthÀs" "Biesel") ("Dominique") "f" _dbb ("HUSBAND" _jpb)("BIRTH YEAR" 1946))
(mp ("BarthÀs") ("Camille" "Xavier") "m" _cxb ("FATHER" _jpb) ("MOTHER" _dbb)
    ("BIRTH YEAR" 1981))
(ms ("BarthÀs") ("Peggy" "Sophie") "f" _psb ("FATHER" _jpb) ("MOTHER" _dbb)
    ("BROTHER" _cxb)("BIRTH YEAR" 1973))
(mp ("BarthÀs") ("Pierre-Xavier") "m" _pxb ("BROTHER" _jpb)("AGE" 54))
(mp ("BarthÀs") ("Claude" "Houzard") "f" _chb ("HUSBAND" _pxb)("AGE" 52))
(mp ("BarthÀs") ("Antoine") "m" _ab ("FATHER" _pxb)("MOTHER" _chb)
    ("COUSIN" _cxb _psb)("AGE" 27))
(mp ("BarthÀs") ("S≫bastien") "m" _sb ("FATHER" _pxb)("MOTHER" _chb)
    ("BROTHER" _ab)("COUSIN" _cxb _psb)("AGE" 24))
(mp ("BarthÀs") ("Emilie") "f" _eb ("FATHER" _pxb)("MOTHER" _chb)
```

```
    ("BROTHER" _ab _sb)("COUSIN" _cxb _psb)("AGE" 21))
(mp ("BarthÀs") ("Papy") "m" _apb ("SON" _jpb _pxb)("AGE" 85))
(mp ("BarthÀs") ("Mamy") "f" _mlb ("SON" _jpb _pxb)("HUSBAND" _apb)("AGE" 81))
(mp ("Labrousse" "BarthÀs") ("Marie-GeneviÀve") "f" _mgl ("FATHER" _apb)
    ("MOTHER" _mlb)("AGE" 48)("BROTHER" _jpb _pxb))
(mp ("Labrousse") ("Michel") "m" _ml ("WIFE" _mgl)("AGE" 52))
(mp ("Labrousse") ("Sylvie") "f" _sl ("FATHER" _ml)("MOTHER" _mgl)
    ("COUSIN" _psb _cxb _ab _sb _eb)("AGE" 19))
(mp ("Labrousse") ("Claire") "f" _cl ("FATHER" _ml)("MOTHER" _mgl)("SISTER" _sl)
    ("COUSIN" _psb _cxb _ab _sb _eb)("AGE" 18))
(mp ("Labrousse") ("AgnÀs") "f" _al ("FATHER" _ml)("MOTHER" _mgl)
    ("SISTER" _sl _cl)("COUSIN" _psb _cxb _ab _sb _eb)("AGE" 15))
(mp ("Canac") ("Bruno") "m" _bc)

;---
(ms ("Shen") ("Weiming") "m" _wms)
(ms ("de Azevedo") ("Hilton") "m" _hda)
(ms ("Scalabrin") ("Edson") "m" _es)
(ms ("Marchand") ("Yannick") "m" _ym)
(ms ("Vandenberghe") ("Ludovic") "m" _lv)
;---
(mp ("Fontaine") ("Dominique") "m" _df)
(mp ("Li") ("ChuMin") "m" _cml)
(mp ("Kassel") ("Gilles") "m" _gk)
(mp ("Gu≫rin") ("Jean-Luc") "m" _jlg)
(mp ("Trigano") ("Philippe") "m" _pt)

(m-definstance ORGANISM ("ABBREVIATION" "UTC")
               ("NAME" "Universit≫ de Technologie de CompiÀgne")
               ("EMPLOYEE" _jpb _dbb _df _gk _pt)
               ("STUDENT" _wms _hda))
(m-definstance ORGANISM  ("ABBREVIATION" "IC")
               ("NAME" "Imperial College")
               ("STUDENT" _psb))

(moss::mformat "~%*** saturating family-links ***")

;(setq *persons* (send *qh* '=process-query '(person)))
(defParameter *persons*
  (list _jpb _dbb _cxb _psb _pxb _chb _ab _sb _eb _apb _mlb _mgl _ml _sl _cl _al _wms
        _hda _es _ym _lv _df _cml _gk _jlg _pt))
(defVar *rule-set* nil)
(setq *rule-set* nil)
(defMacro ar (&rest rule)
  `(push ',rule *rule-set*))
;;;  Rules to close the family links
;;;
;;; If person P1 has-sex M
;;;              has-brother P2
;;;    then person P2 has-brother P1
```

```
(ar "m" has-brother has-brother)
;;;
;;; If person P1 has-sex M
;;;              has-sister P2
;;;    then person P2 has-brother P1
(ar "m" has-sister has-brother)
;;;
;;; If person P1 has-sex M
;;;              has-father P2
;;;    then person P2 has-son P1
(ar "m" has-father has-son)
;;;
;;; If person P1 has-sex M
;;;              has-mother P2
;;;    then person P2 has-son P1
(ar "m" has-mother has-son)
;;;
;;; If person P1 has-cousin P2
;;;    then person P2 has-cousin P1
(ar "m" has-cousin has-cousin)
(ar "f" has-cousin has-cousin)
;;;
;;; If person P1 has-sex M
;;;              has-son P2
;;;    then person P2 has-father P1
(ar "m" has-son has-father)
;;;
;;; If person P1 has-sex M
;;;              has-daughter P2
;;;    then person P2 has-father P1
(ar "m" has-daughter has-father)
;;;
;;; If person P1 has-sex M
;;;              has-wife P2
;;;    then person P2 has-husband P1
(ar "m" has-wife has-husband)
;;;
;;; If person P1 has-sex F
;;;              has-brother P2
;;;    then person P2 has-sister P1
(ar "f" has-brother has-sister)
;;;
;;; If person P1 has-sex F
;;;              has-sister P2
;;;    then person P2 has-sister P1
(ar "f" has-sister has-sister)
;;;
;;; If person P1 has-sex F
;;;              has-father P2
;;;    then person P2 has-daughter P1
```

```
(ar "f" has-father has-daughter)
;;;
;;; If person P1 has-sex F
;;;              has-mother P2
;;;    then person P2 has-daughter P1
(ar "f" has-mother has-daughter)
;;;
;;; If person P1 has-sex F
;;;              has-son P2
;;;    then person P2 has-mother P1
(ar "f" has-son has-mother)
;;;
;;; If person P1 has-sex F
;;;              has-daughter P2
;;;    then person P2 has-mother P1
(ar "f" has-daughter has-mother)
;;;
;;; If person P1 has-sex F
;;;              has-husband P2
;;;    then person P2 has-wife P1
(ar "f" has-husband has-wife)


;;; the version using the MOSS service functions does not seem significantly
;;; faster than the one using the messages

(defun apply-rule (p1 p2 rule)
  "produces a list of messages to send to modify the data.
Arguments:
   p1: person 1
   p2: person 2
   rule: (sex) rule to be applied
Return:
   nil or message to be sent."
  ;; note that we use =get and not =get-id that would require the local id of
  ;; the specified property, e.g. $t-person-name or $t-student-name, which
  ;; cannot be guarantied in a global rule
  (let ((sex (car rule))
        (sp-name (cadr rule))
        (ret-sp-name (caddr rule)))
    (if (and (equal sex
                    (car (moss::%get-value
                           P1
                           (moss::%get-property-id-from-name _jpb 'has-sex))))
             (member P2 (moss::%get-value
                          P1
                          (moss::%get-property-id-from-name P1 sp-name))))
      `(send ',P2 '=add-related-objects ',ret-sp-name ',P1))))

#|
```

```
(defun apply-rule (p1 p2 rule)
  "produces a list of messages to send to modify the data.
Arguments:
   p1: person 1
   p2: person 2
   rule: (sex) rule to be applied
Return:
   nil or message to be sent."
  ;; note that we use =get and not =get-id that would require the local id of
  ;; the specified property, e.g. $t-person-name or $t-student-name, which
  ;; cannot be guarantied in a global rule
  (let ((sex (car rule))
        (sp-name (cadr rule))
        (ret-sp-name (caddr rule)))
    (if (and (equal sex (car (send P1 '=get 'HAS-SEX)))
             (member P2 (send P1 '=get sp-name)))
      `(send ',P2 '=add-sp ',ret-sp-name ',P1))))
|#

#|
(apply-rule _pxb _jpb '("m" HAS-BROTHER HAS-BROTHER))
|#

(defun close-family-links (family-list rule-set)
  "Function that takes a list of persons and computes all the links that must be added
to such persons to close the parent links brother, sister, father, mother, son,
daughter, cousin. It executes a loop until no more changes can occur, applying
rules from a rule set. The result is a list of instructions to be executed."
  (let ((change-flag t) action action-list (count 0))
    (print count)
    ;; Loop until no more rule applies
    (loop
      (unless change-flag (return nil))
      ;; reset flag
      (setq change-flag nil)
      ;; loop on couple of objects in the family-list
      (dolist (P1 family-list)
        (dolist (P2 family-list)
          ;; loop on each rule
          (dolist (rule rule-set)
            ;; if the rule applies it produces an instruction to be executed
            (setq action (apply-rule P1 P2 rule))
            ;; which is appended to the instruction list
            (when (and action
                       (not (member action  action-list :test #'equal)))
              ;; print a mark into the debug trace to tell user we are doing
              ;; some work
              (prin1 '*)
              (incf count)
              (if (eql (mod count 50) 0) (print count)) ; new line after 50 *s
```

```
              (push action action-list)
              ;; in addition we mark that a changed occured by setting the change flag
              (setq change-flag t))))))
    ;; return list of actions
    (reverse action-list)))

(moss::mformat "~%*** closing family links (takes some time) ***")

(defVar instructions)
(setq instructions (close-family-links *persons* *rule-set*))
(mapcar #'eval instructions)

;;; Adding orphans

(m-defobject ("NAME" "Dupond")("sex" "m"))
(m-defobject ("NAME" "Durand" )("SEX" "f")("AGE" 33))

(moss::mformat "~%*** family file loaded ***")
```

Chapter 10

Paths

Contents

10.1 Introduction

When using an ontology and/or a knowledge base, most of the reasoning is done through querying. When the user is interacting by means of natural language, one has to build the queries from the user's input. The input may be noisy when coming from a speech analysis device. The approach developed in this document is to use the structure of the knowledge base for building a formal query from the user's input, i.e. from a list of words.

To do so we use the specific structure of the MOSS representation, and in particular the entry points.

Examples

- From the input:

  ```
  "personne" "prÈsident" "utc"
  ```

  We produce:

  ```
  (($E-PERSON
    ($S-TEACHING-ORGANIZATION-PRESIDENT.OF (> 0)
     ($E-TEACHING-ORGANIZATION ($T-ORGANIZATION-ACRONYM :IS "Utc")))))
  ```

- From the input:

```
"utc"
```

We produce:

```
(($E-UNIVERSITY ($T-ORGANIZATION-ACRONYM :IS "Utc")))
```

Hypotheses

Very little is assumed, however

- we assume that the target concept is the first one in the sentence;
- the series of words in input are extracted from the original input and correspond to terms designing concepts, properties or entry points. Empty words or words not understood are discarded.

10.2 Algorithm

10.2.1 Global Approach

The main idea of the algorithm is to mark the concepts, properties and entry points in the graph representing the knowledge base and to produce a series of paths that includes the marked items in the order they have been given. The approach is overly simple but nevertheless efficient in many cases.

10.2.2 Algorithm

We proceed in several steps:

- First the words are transformed into class-ids, relation-ids, or entry-points. Thus, we obtain an ordered list of places (nodes, relations) that the path must traverse to obtain the formal query.

  Entry points have a different role: they contain a terminal property an operator, and the class to which it may apply. If the class is missing from the original marked items, then it can be recovered from the entry point frame.

- Second, we solve special cases directly. E.g. when there is a single item, or a single entry point frame...
- The main algorithm is to build paths starting from a specific class, the target, that will go through all the items of the ordered list. At each step, we check if we came to the next item of the ordered list in one of the paths. If so, we throw away all the other possibilities, betting on the shortest path heuristic. We then proceed from there. If we did not come to the next item, then a number of path are generated using the relations and inverse relations attached to the class.

To avoid fanning, we limit the possible length of a path, quitting when it is reached.

10.2.3 Processing the Initial Input

The input sentence (list of words) is processed in several steps.

Screening Relevant Terms

The initial input consists of a list of words (strings) from which punctuation marks have been removed, e.g.

```
("quel" "est" "le" "numÈro" "de" "tÈlÈphone" "du"  "prÈsident" "de" "l" "utc")
```

A first function, find-best-entries, takes the list as an input and outputs two lists;

- a list of entry points corresponding to ontology or knowledge base items
- a list of unrecognized words

On the previous example;

```
(HAS-NUMERO TELEPHONE HAS-PRESIDENT UTC)
("quel" "est" "le" "de" "du" "de" "l")
```

Building an Initial List

From the previous list of relevant entry points we replace with internal ids for MOSS objects and class frames for knowledge base entry points. This is done by the function path-process-word-list. The function returns two lists:

- the translation;
- a list of unused words or words not understood.

A class frame is a list containing a concept id and an attribute clause, e.g.

```
($E-UNIVERSITY ($T-ORGANIZATION-ACRONYM :IS "Utc"))
```

The expression is built from the entry point object. In the example: "Utc"

Examples

```
? (moss::path-process-word-list
   '("quelle" "est" "l""adresse" "privÈe" "du" "prÈsident" "de" "l" "utc"))
($E-HOME-ADDRESS $S-PRESIDENT ($E-UNIVERSITY ($T-ORGANIZATION-ACRONYM :IS "Utc")))
("quelle" "est" "l" "du" "de" "l")

? (moss::path-process-word-list
   '("quel" "est" "le" "numÈro" "de" "tÈlÈphone" "du" "prÈsident" "de" "l" "utc"))
($E-PHONE $S-PRESIDENT ($E-UNIVERSITY ($T-ORGANIZATION-ACRONYM :IS "Utc")))
("quel" "est" "le" "de" "du" "de" "l")

? (moss::path-process-word-list
   '("quel" "est" "le" "numÈro" "de" "portable" "du" "prÈsident" "de" "l" "utc"))
($E-CELL-PHONE $S-PRESIDENT ($E-UNIVERSITY ($T-ORGANIZATION-ACRONYM :IS "Utc")))
("quel" "est" "le" "de" "du" "de" "l")
```

10.2.4 Special Cases

Next step is to process special cases. We have several special cases:

- input is a single term corresponding to a class, we return it as a list;
- input is a single entry point, we return it as a list;
- input is a single object relation, we do not process it yet.

10.2.5 Building the Path

In the general case we do the following:

1. We first remove from the list all leading ids that do not correspond to a concept in order to determine a target concept (it may be a class frame). If the remaining list contains a single element, we are done.

2. We fuse adjacent class frame corresponding to the same class. If the remaining list contains a single element, we are done.

3. We navigate from the first concept until we reach the next or until the path becomes too long. To do so, we fan out from the end of the current paths through the neighbors of the current class, actually building a spanning tree. We repeat the process until we hit a relation with the same name as the target relation, or a class that contains the target class. In that case we only keep the branches that end up on the target item, the rationale being that we favor shortest paths.

4. Then we build a set of formal queries. For simple question, the resulting list contains usually a single query.

10.2.6 Results

Some results are shown in the following examples.

Examples

```
? (moss::path-build-query-list '( "utc"))
(($E-UNIVERSITY ($T-ORGANIZATION-ACRONYM :IS "Utc")))

? (moss::path-build-query-list '( "personne"))
(($E-PERSON))

? (moss::path-build-query-list '( #+MCL "tÈlÈphone" #-MCL "t≫l≫phone" "utc"))
(($E-PHONE
  ($S-ORGANIZATION-PHONE.OF (> 0)
       ($E-ORGANIZATION ($T-ORGANIZATION-ACRONYM :IS "Utc")))))

? (moss::path-build-query-list '("personne" "prÈsident" "utc"))
(($E-PERSON
  ($S-TEACHING-ORGANIZATION-PRESIDENT.OF (> 0)
   ($E-TEACHING-ORGANIZATION ($T-ORGANIZATION-ACRONYM :IS "Utc")))))

? (moss::path-build-query-list '( "tÈlÈphone"  "prÈsident"  "utc"))
(($E-PHONE
  ($S-PERSON-PHONE.OF (> 0)
   ($E-PERSON
    ($S-TEACHING-ORGANIZATION-PRESIDENT.OF (> 0)
      ($E-TEACHING-ORGANIZATION ($T-ORGANIZATION-ACRONYM :IS "Utc")))))))
```

The resulting list of queries can then be used by the calling program to access entities corresponding to the queries.

10.3 Conclusion

This simple set of functions is useful in the case of an agent receiving a natural language input. It can use the them to build formal queries to access its own knowledge base, trying to make sense of the input.

Of course there are limitations to the approach, namely:

- when the input is so noisy that terms are missing or not understood (which happens with vocal inputs);
- when the path contains the same class to be traversed several times;
- when some of the crucial data does not have corresponding entry points (e.g. an unknown family name)
- when the path is too long or too complex.

However, in general as a first attempt, the functions presented here are quite useful.

Chapter 11

Persistency

Contents

11.1 Introduction

MOSS is a knowledge representation language, well suited for developing ontologies. In order to develop long term applications, one needs persistency, i.e. to save ontologies on disk in an efficient fashion. Persistency can be achieved in different ways:

- saving everything into a flat file from time to time and reloading the file as needed;

- saving objects into a relational database like MYSQL;
- saving objects into an object database.

The first solution is valid when the knowledge base is small. However, it does not scale up to large knowledge bases in particular when they change slowly.

The second solution requires an (easy to do) interface to the relational database. However, a relational database is a rigid environment that cannot be modified easily. It cannot cope with dynamically changing classes or objects.

The third solution is ideally suited to save an agent ontology and knowledge base as was demonstrated by the MLF object base that led to the commercial MATISSE product. Lisp environments offer the possibility to save Lisp object persistently, which was the adopted solution.

11.2 Overall Approach

The approach differs whether one uses MCL in Mac OSX or ACL in Windows XP. However, the overall approach is extremely simple. Each MOSS object is a Lisp a-list and is the value of symbol acting as an internal identifier, e.g. $E-PERSON.234. Thus the object base is organized as a direct access file using the object identifiers as keys.

Mac OS X: In the MCL environment (Mac) we use the WOOD's object store, declaring the data base as a persistent hash table.

Windows XP: In the ACL environment (Windows) we use Allegrocache, an object store proposed by Franz Lisp, for similar purposes.

In the two approaches, the main design choice for the decision when to save data, was left to the user, contrary to the decision of letting the system do the saving automatically. The reason is that if the computer crashes, then the data saved into the file may be inconsistent, and it is difficult to repair them.

Two problems pop up:

- do we save MOSS system objects into the agent data store?
- how do we cope with temporary inconsistency between the data in core and the data on disk?

The first point concerns MOSS system objects, e.g. meta-classes, some properties and methods. It would be simpler to save them with the rest of the objects. However, when a new version of MOSS is loading, system objects have to be updated, which is not an easy task, due to interconnections between system objects and application obejcts (in particular through entry-points). If we do not save system objects into the data store, then we have to remove system information before saving an application object. In that case when we restart MOSS, we recreate the system objects and when we open the application database we reconnect MOSS system objects with the application objects.

The second point concerns changes in a class or an object. When editing, changes will be done in core and several objects will be modified (e.g. backpointers in inverted properties). We cannot do a disk access each time a small change is performed. Thus the solution is to modify the object in core and at the same time record the objects that have been modified. When we are ready to update the saved data, we then record the objects that have been modified and need saving.

We examine the two problems in the following sections.

11.3 System Objects

System objects refer to all the objects that are created when loading the MOSS files. They include all the system classes defining the structure of MOSS objects and the corresponding methods[1]. Such objects and methods are defined by defmossxxx macros in the various MOSS files. System objects are not supposed to be modified by the user. They are defined in the "MOSS" name-space (:moss package), thus, it should be easy to separate them from the application objects that live in their own name spaces. However, this is not so simple, mainly because of entry-points.

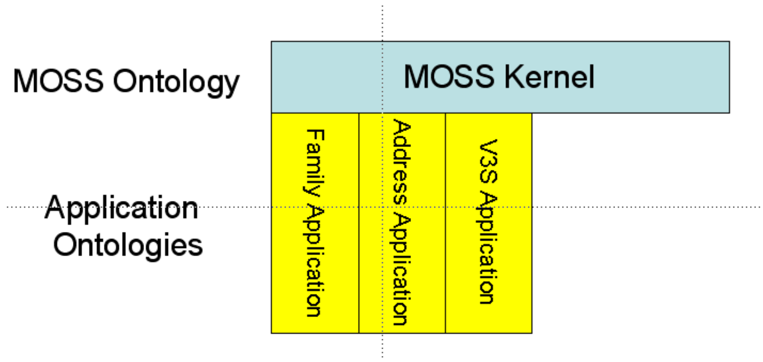

Figure 11.1: Structure of MOSS ontologies

In practice the :moss name space and the application name space are distinct (Fig.11.1). However, if care is not taken there may be many cross references, like

- when an object is built, all properties from the :moss space can be used
- MOSS entry points are global exported symbols that may refer to objects located in various name spaces

The global policy is to separate as much as possible the :moss space from the application name space, by creating application objects acting as proxies for the MOSS objects, namely:

- a model of moss-system ($E-SYS), counter ($E-SYS.CTR) and entry-point (e.g. ONTOLOGY-FAMILY-MOSS)
- an instance of moss-system ($E-SYS.1) and its entry point (e.g. ONTOLOGY-FAMILY-MOSS-SYSTEM)
- a counter class ($E-CTR), and entry point (COUNTER)
- a method class ($E-FN), counter ($E-FN.CTR) and entry point (METHOD) for instance and own methods
- a universal method class ($E-UNI), counter ($E-UNI.CTR) and entry-point (UNIVERSAL-CLASS)
- a NULL-CLASS (*NONE*) and counter (*NONE*.CTR) used for orphans, and entry point (NULL-CLASS)
- a universal class (*ANY*) and entry point (UNIVERSAL-CLASS)

[1]Methods are also objects.

11.3.1 Entry-Points

Entry-points are used to access an object directly, e.g. the name of a person, the name of a concept, the name of a property, etc. A system class may have the same name as an application concept. Take for example the name "CONCEPT." It is the name of the MOSS metaclass; but may also refer to a term in any application. Therefore, the entry-point with key CONCEPT is an object linked to both a system object and to an application concept. When saving an entry-point object into the application database, it must be stripped of the system references. Conversely, when opening a new application, the system entry-points must be augmented with the application data.

Application entry-points are saved in the application environment (referred to by *moss-system* or *ontology*), system entry-points are saved in the system environment in the :moss package. Shared entry points are saved with the system. Thus, to obtain all the entry points accessible within an application, both application and system environments must be inspected and fused.

11.3.2 Methods

Methods are first class objects. However, at some stage, I decided to separate the method code from the object, giving the function implementing the method a unique identifier. The reason was that we could then change the code of the function without changing the method object. This mechanism is of course also valid for applications, which means that saving a method object requires saving the corresponding function. Conversely, when loading a method the associated function must be loaded and compiled at the same time.

11.3.3 Properties

Properties are defined using generic properties. When a property is created, the corresponding generic property is also created. Generic properties should be saved and loaded along with each property.

11.3.4 Cross-Linking

Cross-linking refers to the possibility of linking an application object to a system object, e.g. be declaring an application class as a subconcept of a system concept (class). If we save the application class separately from the system class, then we have a pointer pointing to the system class that may change in the following release of the system. On the system side, the objects are not saved but regenerated when loading the system. Thus, when reloading the application, we have lost the backpointer to the application object. A special mechanism should be designed to restore such lost links, when opening and closing an application, **which currently is not done**.

11.3.5 Functions

All functions defined to work with the ontology need to be saved (and restored), unless they are part of files that are loaded separately.

11.4 Editing Objects

Usually an ontology is created in a text file that is loaded into the system. The concepts, properties, methods and associated knowledge base are then stored into the database. Doing so creates two problems related to comments and to the link with the original text file.

11.4.1 Comments

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

11.4.2 Edited objects

Once objects are edited and saved into the database, we loose the connection with the text file that may have been used to create the objects to start with.

11.5 Transactions

Editing an object leads to making many changes in other objects (e.g., neighbors, entry-points). All modified objects must be marked and updated on disk in a single operation when the user decides so.

To do well one should first save an object to be modified into a temporary file, then make the changes, and if the transaction was successful discard the temporary file.

11.6 Loading and Saving Ontologies

Loading and saving ontologies can be done through the MOSS window interface or by programming.

11.6.1 Using the MOSS Window Interface

Loading the Initial Ontology

This is done by writing the name of the ontology in the Application slot of the window and pushing the LOAD FILE button (Fig.11.2).

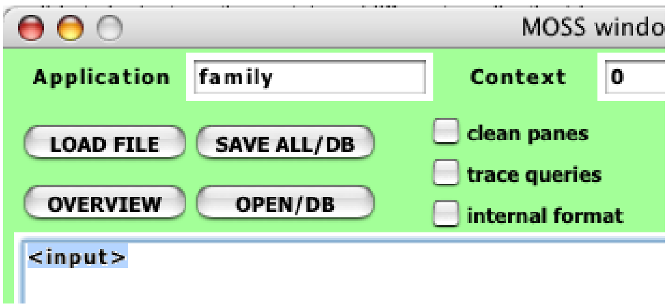

Figure 11.2: Loading the *family-m* application ontology

After the ontology is loaded the window operates in the ontology package (with the family-m example, this would be the :family package).

Creating the Database and Saving the Ontology

The simplest way to do it is to click the SAVE ALL/DB button (Fig.11.2). This will create a Wood database contained in a file of the application folder. In our example the file is named family-m.odb in the family application folder. The application is now persistent.

Starting a New Session

When starting a new session after having restarted MOSS, one can recall an application by writing the name of the application into the application slot of the MOSS window, then clicking the OPEN/DB button. MOSS will open the database, reconnect all system information and switch to the application package (in our example the :family package).

Examining another Ontology

It is possible to look at another ontology (different application) by repeating the steps of the previous section. Doing that will close the current database, disconnect the MOSS ontology, open the new application database, reconnect the MOSS ontology and set the package to the new ontology package.

Going back to the previous situation requires redoing the same steps.

11.6.2 Programming Approach

This section describes what can be done through text programming. Programming is demonstrated in the Lisp listener, the MOSS window being closed[2].

First Time Saving

We assume that a new ontology has been created in its own package, either directly or by loading a text file, *and that we are in this ontology package* (known as the application package).

First we must create a database:

```
? (db-create :filename "my-database" :package-key :family)

#<IO WOOD:PHEAP to
#P"Odin:Users:barthes:MCL:OMAS-MOSS 8.0:MOSS:applications:my-database.odb">
```

This creates a file named my-database.odb in the applications sub-directory of the MOSS folder. If the filename is not specified, then the file will be called MOSS-ONTOLOGIES.odb. If a file-pathname is provided, MOSS will use the pathname. Note that several ontologies can be saved into the same object store. package-key is a compulsory argument and must be the key of the ontology package.

We can check if the ontology stub exists:

```
? *ontology*
$E-SYS.1

? $E-SYS.1
((MOSS::$TYPE (0 $E-SYS)) (MOSS::$ID (0 $E-SYS.1))
 (MOSS::$SNAM (0 (:EN "FAMILY Test Ontology MOSS")))
 (MOSS::$PRFX (0 FAMILY-TEST-ONTOLOGY-MOSS))
 (MOSS::$EPLS
  (0 FAMILY-TEST-ONTOLOGY-MOSS FAMILY-TEST-ONTOLOGY-MOSS-SYSTEM METHOD
   UNIVERSAL-METHOD COUNTER NULL-CLASS UNIVERSAL-CLASS PERSON PERSONNE HAS-NAME
...
   $S-ORGANIZATION-EMPLOYEE $S-STUDENT $S-ORGANIZATION-STUDENT)))
```

Now we save the world by sending the =save message:

[2]Otherwise messages are forwarded to the MOSS window.

```
? (send *ontology* '=save)
:OBJECTS-SAVED
```

All objects are accessible through the *ontology* object. MOSS objects are removed as well as system back pointers from entry points. Then, the whole lot is saved into the disk cache one object at a time. The database file pathname is kept internally by MOSS after the creation of the database.

We can now close the database. However, this will be done automatically when exiting the Lisp environment.

```
? (db-close)
:DONE
```

Creating or Modifying Ontology or KB Objects

When creating or modifying objects of the ontology or the KB, the object base should be updated. The simplest way to do it is to keep a list of the objects that have been modified, and when the creation or modification is finished, to update the object base by overwriting existing objects. A specific slot of the agent structure indicates if a transaction in on-going or if each created or modified object must be saved without delay.

Opening an Already Existing Database

When restarting MOSS and closing the MOSS window, an already existing database can be opened as follows:

```
? (db-open "my-database")
;*** my-database database opened ***
#<IO WOOD:PHEAP to
#P"Odin:Users:barthes:MCL:OMAS-MOSS 8.0:MOSS:applications:my-database.odb">
```

The MOSS system executes now in the applications package:

```
? (db-load '*ontology*)
$E-SYS.1

? (db-load '$E-STUDENT)
((MOSS::$TYPE (0 MOSS::$ENT)) (MOSS::$ID (0 $E-STUDENT))
 (MOSS::$ENAM (0 (:EN "STUDENT" :FR "ETUDIANT"))) (MOSS::$RDX (0 $E-STUDENT))
 (MOSS::$ENLS.OF (0 $E-SYS.1)) (MOSS::$CTRS (0 $E-STUDENT.CTR))
 (MOSS::$DOCT
  (0 (:EN "A STUDENT is a person usually enrolled in higher education")))
 (MOSS::$IS-A (0 $E-PERSON)) (MOSS::$PS (0 $S-STUDENT-COURSES))
 (MOSS::$SUC.OF (0 $S-ORGANIZATION-STUDENT)))
```

Note: Closing the database does not change the execution package.

Operations on the Database

The possible operations on the database are:

- db-close, closes the database, disconnecting MOSS
- db-load *key*, loads the value associated with a key from the database

- db-open *filename* or *(choose-file-dialog)*, opens a specific database
- db-store *key value*, stores the key and value (key must be a symbol)
- db-vomit *package-key*, prints all the keys in the database, corresponding to the package-key area

Note that after closing a database, we remain in the ontology package (application package).

11.7 Implementation (Version 0)

Several implementations are possible. We start with a simple one.

11.7.1 Overall Approach

Here we try to keep things as simple as possible, making the following assumptions:

1. Ontology concepts, KB objects and related functions (methods) are saved into the object base.
2. The first time around, the ontology and KB are created in a text file, loaded and saved in a bulk mode.
3. All new concepts and KB instances are created by program, i.e. they are not created or modified through the MOSS editor.
4. Objects are loaded as needed[3] by the QUERY mechanism, or by some kernel methods[4].

This has the following consequences:

1. When saving application objects, entry-points should be stripped of the system items.
2. After reloading the system, system entry points should be added to data.
3. Saving a method should save the corresponding function.
4. Loading a method should load the corresponding function.
5. Saving a property should save the generic property.
6. Loading a property should load the generic property.

11.7.2 Saving Objects

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

- concepts are saved normally with the exception of omas-agent and omas-skills[5]
- properties, entry points are saved normally, generic properties are checked for existence (this is probably not necessary)
- methods are saved together with corresponding functions
- nothing special for individuals
- macros and functions, must be saved as text. They are saved on the list moss::*moss-system*.

[3]When they are not in core.
[4]In practice by service functions called by the kernel methods.
[5]This can be done easily by giving the concept and properties a =save method that does nothing.

11.7.3 Objects Not to Be Saved

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

11.7.4 Opening a Database and Reconnecting MOSS

When an application database is opened, MOSS does several things:

1. It loads the local *moss-system* stub.
2. It reconnects the various entities from the stub to the MOSS counterparts.
3. It loads and initializes application global variables.
4. It loads and initializes application functions.
5. It fuses entry-points common to the application and to MOSS.
6. It loads the application concepts, methods, attributes, relations, and inverse properties.

11.7.5 Multiple Ontologies and Object Stores

The implementation of object stores (MCL Wood with Mac, and AllegroStore for ACL on Windows) only allows opening a single database for each Lisp environment. Consequently, when we have several parallel ontologies, we must partition the object store in such a way as to keep the objects from being intermingled. Thus, partitions have been done *on the basis of the ontology package*.

In both cases (MCL and ACL)[7] we create a database with a root that contains a list of hash tables. Each hash table is associated with a package associated with a particular ontology[8]. The storing and loading functions check the package of the key symbol to determine where to store or to look for the key.

Since we have a single database, accesses to the database from different threads must be done independently, which is the case when each agent uses a different partition. There is a risk however that a thread starts a write (commit), is interrupted and another thread starts a second write. Such a risk is very small and is neglected, meaning that we commit every single write, which may be expensive.

11.8 Possible Improvements

11.8.1 Minor Necessary Improvements

Several improvements could be done:

- Currently Lisp application macros are not allowed in the database file, which prevents from using them in text files and subsequently in application programs.
- Accessor functions (HAS-XXX) attached to properties are not saved, which prevents from using them in methods like =summary.
- The question related to the version and context of the application should be addressed. Currently we assume that context is 0, and when opened the database inherits the current context.

[6] One must be careful to cook up unique entry points for such objects. Otherwise if the entry point is shared and not saved, this will introduce an inconsistency in the database.

[7] Both approaches (MCL and ACL) are very similar.

[8] In the case of an agent it will be the agent package.

11.8.2 Major Improvements

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

Chapter 12

Editor

Contents

12.1 The MOSS Editor

(Fig.12.1) displays the MOSS editor window. It is a fairly complex interface allowing to edit any kind of object. However, the objects being edited are kept in memory and if they were loaded from a text file, the original text file remains unchanged. This limits the scope of the editor in this case. The editor is only useful only when the objects are kept in a persistent object base.

12.1.1 The Editing Process

During edition, objects are modified, new objects may be created and existing objects may be destroyed. The process takes place within a transaction, meaning that it may be committed or aborted. This is done by clicking the `commit` or `abort` button at the right top corner. Whenever the process is aborted, all operations are discarded and the system is returned to its state prior to starting the editing process.

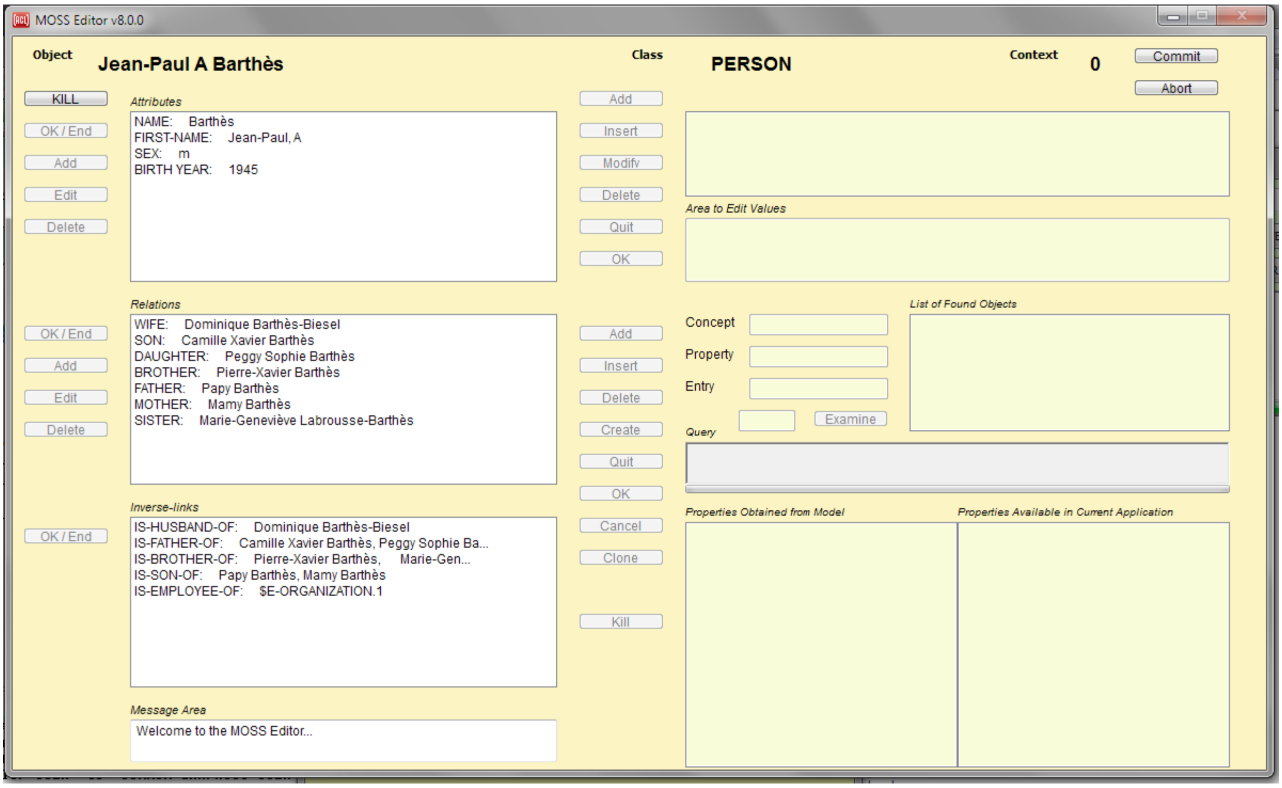

Figure 12.1: The MOSS editor window

12.1.2 The Editing Window

The top part of the window displays the reference of the object being edited (as the result of applying the =summary method), the name of the corresponding concept, the context number, and the two buttons commit and abort.

A special mention should be done of the top left button, kill, used to destroy the current object (on double-click as a safety measure).

The left part of the window is devoted to the display of the selected object. The top left part shows the attributes, the middle part the relations, the bottom part the inverse relations. At the very bottom a special area is used to display (error) messages. The right part of the window is used to edit the object, i.e., add values, delete values, add properties, or link the objects to other objects.

When the window first appears all buttons are dimmed except for kill, commit and abort. I.e., nothing can be done. Some buttons are activated when some part of the object is selected in one of the left windows. For example the left top buttons become active when we select the first-name property (Fig.12.2). Note that the other left windows become dimmed, telling us that we are focusing on editing an attribute

To reset the display to the initial state, we can click the OK button.

The following paragraphs explain what happens for each of the different areas.

12.2 Editing an Attribute

The various possibilities for editing an attribute are:

- Display the list of values (by clicking or double-clicking the attribute)

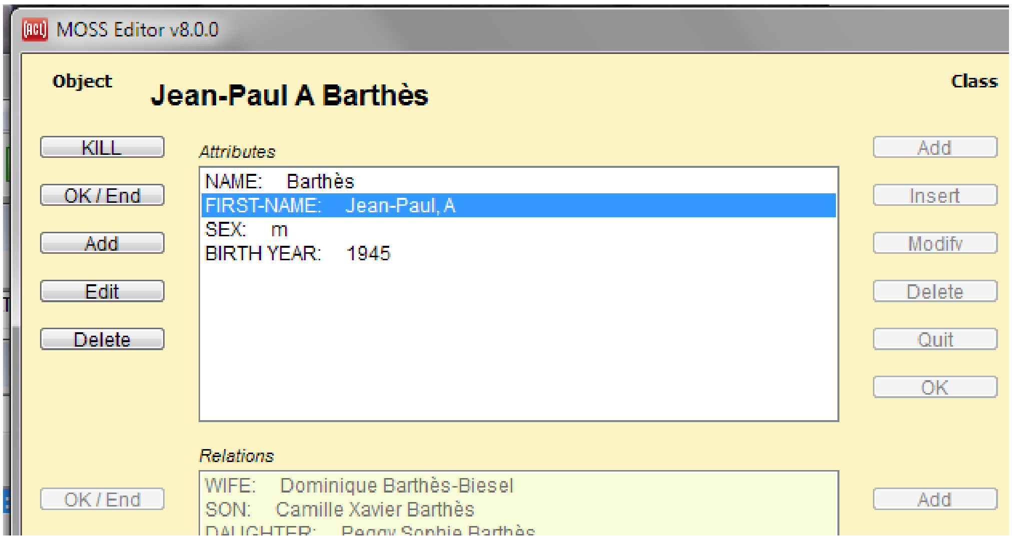

Figure 12.2: Selecting an attribute

- Add new values (by clicking the ADD button)
- Edit the set of values (by clicking the EDIT button)
- Delete some values (by clicking the DELETE button)

12.2.1 Displaying Values of an Attribute

When an attribute is selected in the attribute window (top left), then the corresponding line is marked as selected and the left buttons are activated.

When the data is multiple, then double-clicking on the data displays a small table with one entry per line (Fig.12.3). If the data is a single value, nothing happens.

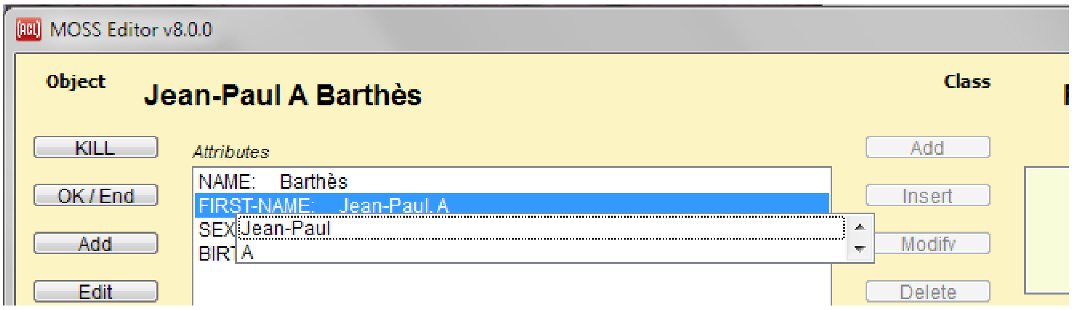

Figure 12.3: Displaying attribute values

When the shift key is pressed while double-clicking on a selected attribute, then entry-points (e.g. when there are synonyms) are shown for attributes having entry points. For example the entry points associated with the value <Barthès> of property NAME is the list (BARTHÈS)(Fig.12.4).

When the attribute has no entry-point, which is the case for attributes FIRST-NAME, SEX, AGE, then double-clicking while holding the command key yields a beep sound and the message

```
No entry point method for this property
```

appears in the message window.

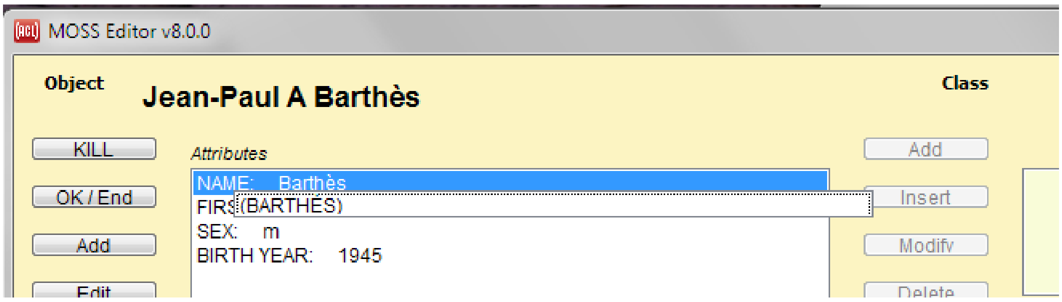

Figure 12.4: Displaying entry points

12.2.2 Adding a New Attribute

Whenever the left Add button is active, clicking it will ask MOSS to add a new attribute to the object. Two cases occur, depending on whether the edited object is an instance of a class or an orphan (classless object):

- When the edited object is an instance, the list of attributes recorded at the class level appears in the "*Properties Obtained from Model*" window (left side of the bottom right area - Fig.12.5), and the list of other attributes present in the system appear in the right window, labeled "*Properties Available in Current Application.*
- When the edited object is an orphan, then only the right list appears.

Thus the user can add an attribute from the model list, e.g. BIRTH-YEAR or NICK-NAME, or decide to add another attribute present in the system from the general attribute list, e.g., LABEL. This means that *MOSS objects can have properties that are not part of their class description*, which is somewhat different from standard object systems.

Note that each property is followed by the class (concept) for which it has been defined.

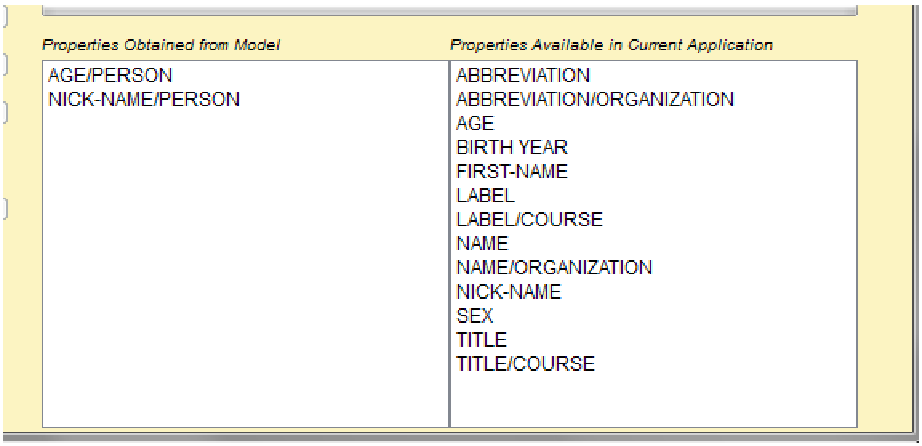

Figure 12.5: Attributes that can be added

Once the attribute has been selected by clicking on it, then the attribute selection area dims and the attribute editing area (top right part) is activated. The name of the selected attribute shows up on top of the first window, and the cursor is positioned in the second window. The OK button is activated (Fig.12.6). The following message appears in the message area

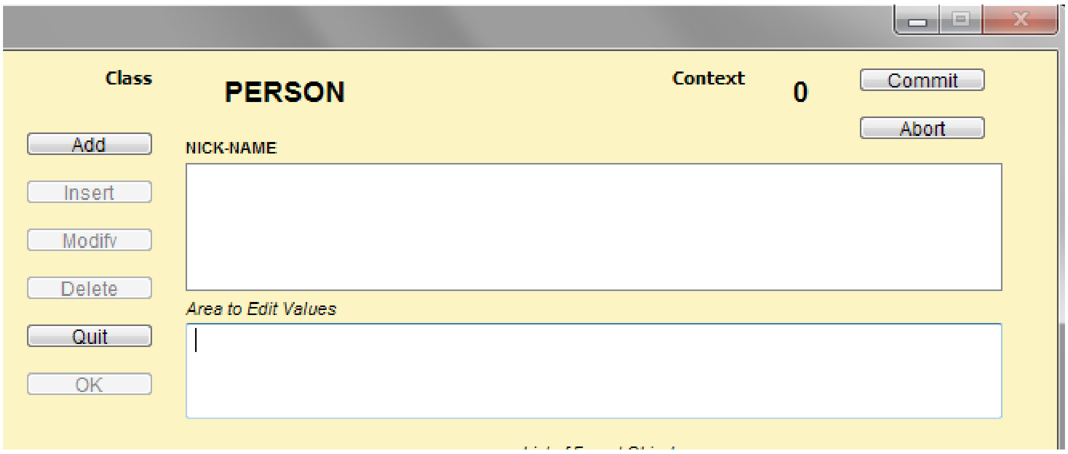

Figure 12.6: Ready to add an attribute

`You can now add a value for the attribute.`

Once the value attached to the property has been added, then we click the `add` button and the entered value appears in the top right window as the first value attached to the NICK-NAME attribute (Fig.12.7). We are now ready to enter more values. Remember MOSS attributes are multi-valued and the values are ordered for the attributes.

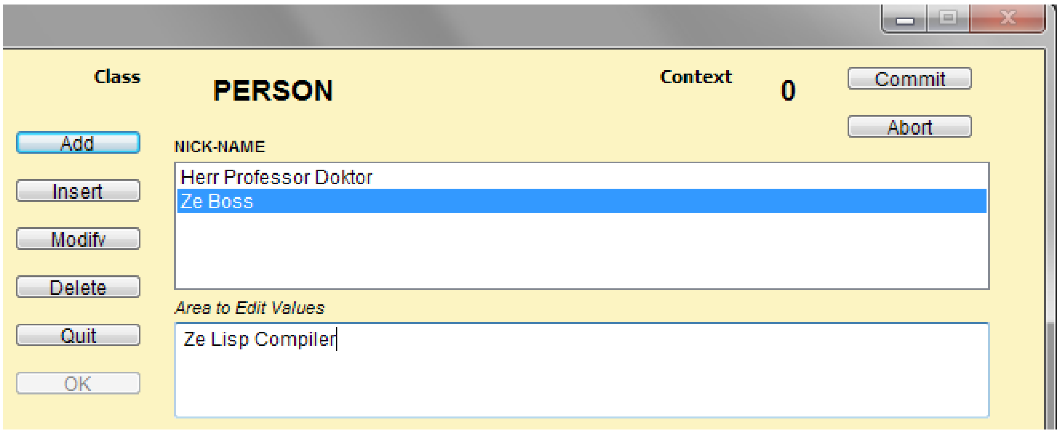

Figure 12.7: Adding a value for an attribute

Note that at the same time the left window displaying the object attributes is updated as well.

When no more values are needed, then we can return to doing other things by clicking into the top left window displaying the object attributes. All areas but the attribute area are dimmed.

Note Clicking the `add` button while no value has been typed in adds an empty value. A warning message appears in the bottom right window entitled *Message Area*:

`You just added an empty value.`

An indication of an empty value is given by the presence of a comma in the attribute list in the *Attributes* area and as a blank line in the list of values (top right window). An empty value can be selected in the list of values in the right top window, and deleted.

12.2.3 Modifying the Values Attached to an Attribute (Right Top Area)

Modifying the values attached to an attribute is done readily. For example, if we want to modify the recently created NICK-NAME attribute, then we select the corresponding line in the left attribute window and the same situation as in Fig.12.7 occurs with the exception that the OK button is not active. However when we click into a blank area of the detailed window then the OK button becomes active and the following message appears in the message area:

`No selected value. You can only add more values.`

We can then add values as done previously.

When we select an existing value however, then the Add Insert Delete Modify buttons become active, with the following effects:

- **Add**. Clicking the Add button allows us to enter a value that will appear right *after* the selected value or at the end of the list of nothing is selected.
- **Insert**. Clicking the Insert button allows us to enter a value that will appear right *before* the selected value.
- **Delete**. Clicking the Delete button deletes the selected attribute. No confirmation is asked.
- **Modify**. Clicking the Modify button allows to edit the selected value, and then replace the old one at the same position.

All the changes are also posted in the attribute display area in the top left hand-side of the edit window. At any time, clicking in this window terminates the editing.

12.2.4 Deleting a Given Attribute

Clicking the Delete button remove the selected value and all attached values from the object. No confirmation is asked.

12.2.5 Re-enabling the Other Areas

After the attribute edition is finished, we are back to a situation like Fig.12.2 with all areas dimmed except for the attribute area. Clicking in the left area tell the system to leave the attribute edit area.

12.3 Editing a Relation

The various possibilities for editing an attribute are:

- Display (by clicking or double-clicking on the attribute)
- Add (by clicking the ADD button)
- Edit (by clicking the EDIT button)
- Delete (by clicking the DELETE button)

12.3.1 Displaying a Relation

When a relation is selected in the relation window (center left), then the corresponding line is marked as selected and the left buttons are activated.

Double-clicking on the data displays a local table with one entry per line (Fig.12.8). Clicking on one of the entries of the table opens a new editor window with the corresponding object to edit.

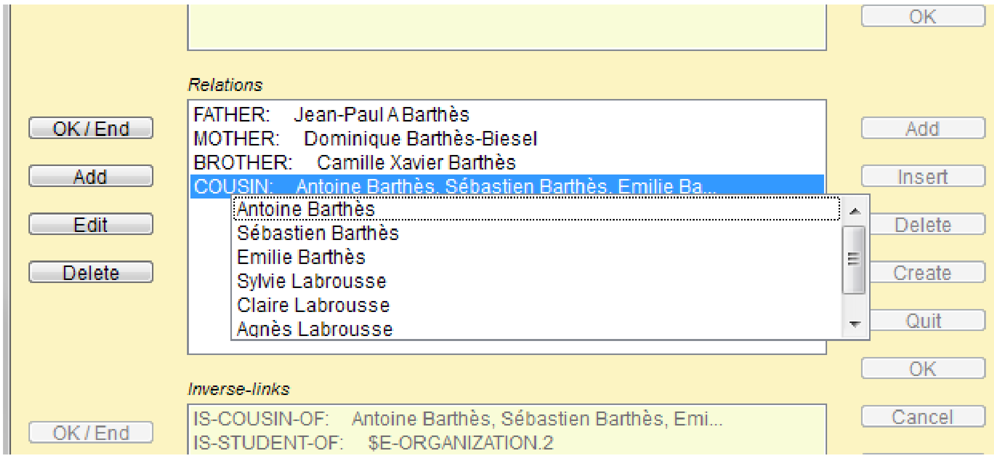

Figure 12.8: Displaying details of the COUSIN relation

12.3.2 Adding a New Relation

Whenever the left Add button is active, clicking it will ask MOSS to add a new relation to the object. Two cases occur, depending on whether the edited object is an instance of a class or an orphan (classless object):

- When the edited object is an instance, the list of relations recorded at the class level appears in the *Properties Obtained from Model* window (left side of the bottom right area - Fig.12.9), and the list of other relations present in the system appear in the right window, labeled *Properties Available in Current Application.*
- When the edited object is an orphan, then only the right list appears.

Thus, the user can add a relation from the model list, e.g. COUSIN or NEPHEW, or decide to add another relation present in the system from the general attribute list, e.g., COUNTER. This means that *MOSS objects can have properties that are not part of their class description*, which is somewhat different from standard object systems.

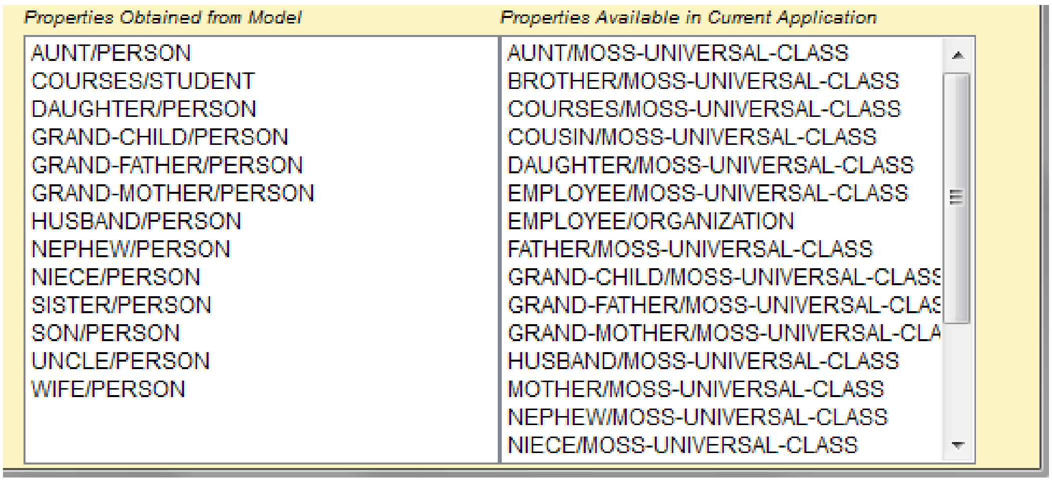

Figure 12.9: List of relations that can be added to a person

Once the relation has been selected by clicking on it, then the relation selection area dims and the relation editing area (middle right part) is activated. The name of the selected relation shows up on top of the first window, and the cursor is positioned in the second window. The Cancel button is activated (Fig.12.10). The following message appears in the message area:

```
You can now search for an object to add, or specify a concept.
```

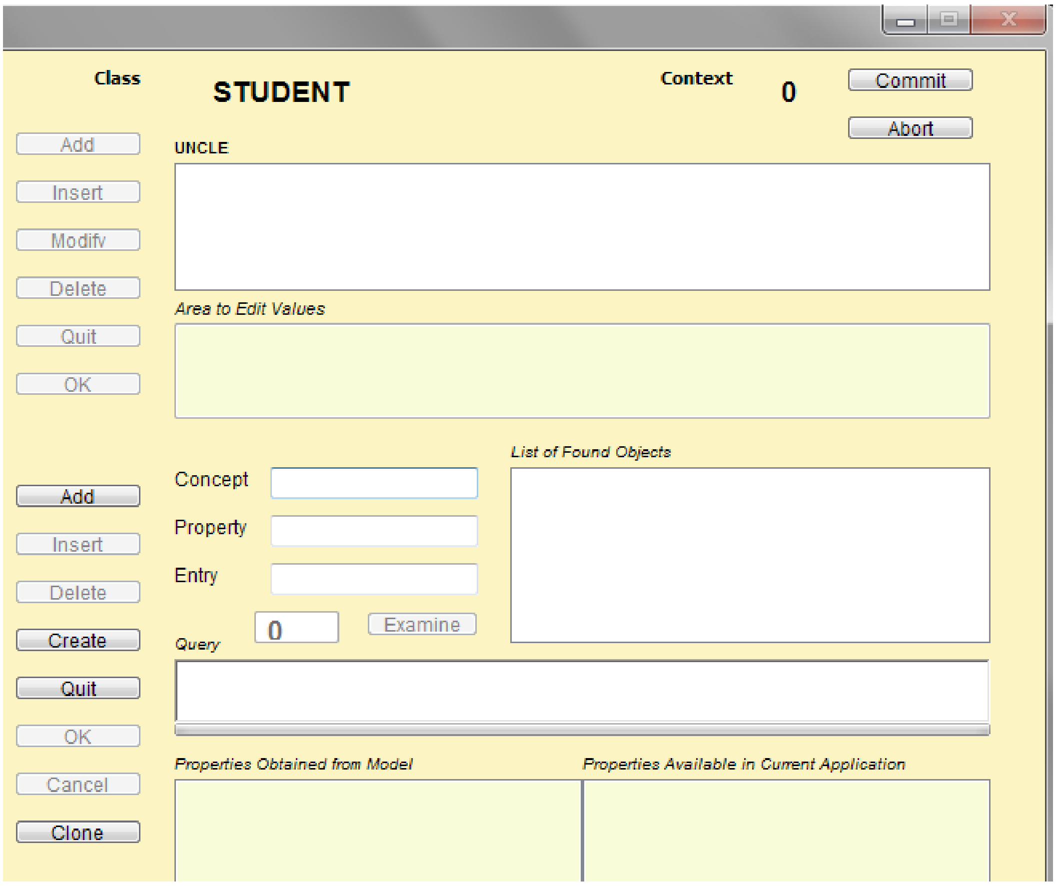

Figure 12.10: Adding a new relation

In the top right hand-side part of the editor window the name of the concerned relation appears (here UNCLE). The center part of the window is used to locate objects to connect to the edited object. When objects exist they must be found. When they do not exist they can be created.

The bottom part of the area can be used to input queries for locating objects.

The right center part of the editor window helps to locate objects.

For example, if we know an entry point (here the family name "canac"), we can use it to locate objects, as shown Fig.12.11.

We can add an object by selecting it (Fig.12.11) and clicking the Add button. The object appears in the top right area and the relation and new object appear in the left list of relations.

Clicking the Cancel button deselects all selected objects.

If we look for a person, we can specify the class name followed by Return or Enter, as shown Fig.12.12. The count figure gives the name of available persons. The Examine button allows to list them as shown Fig.12.13.

Objects can then be selected from the list and added using the add button. Multiple selections are permitted. Clicking Cancel deselects all selected objects.

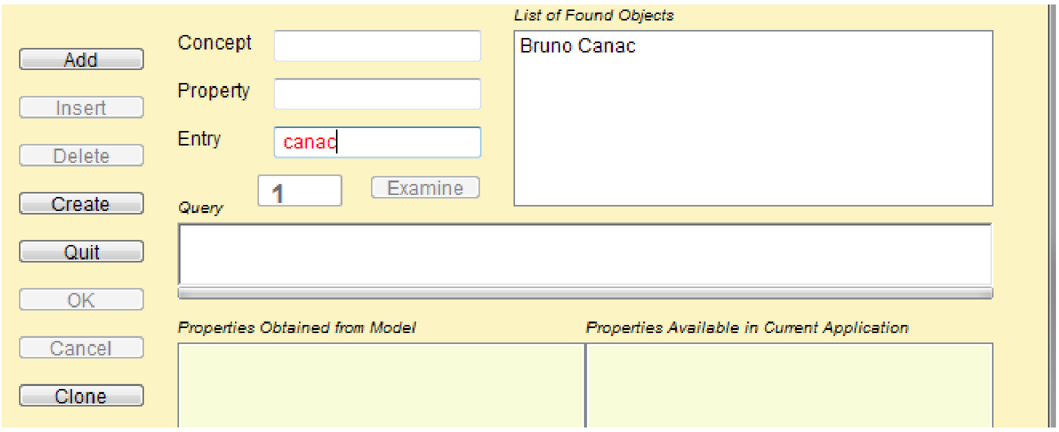

Figure 12.11: Locating an object to link from its entry point (here CANAC in the value slot)

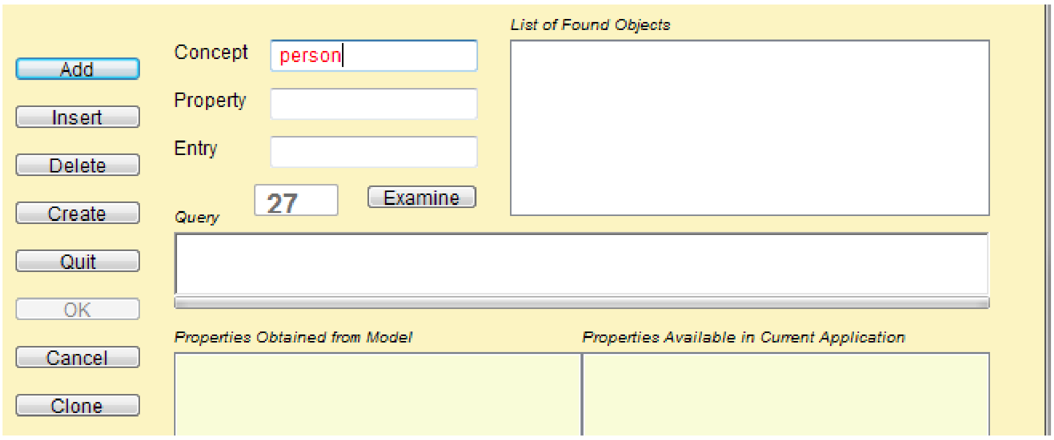

Figure 12.12: Locating objects of a given class (here PERSON)

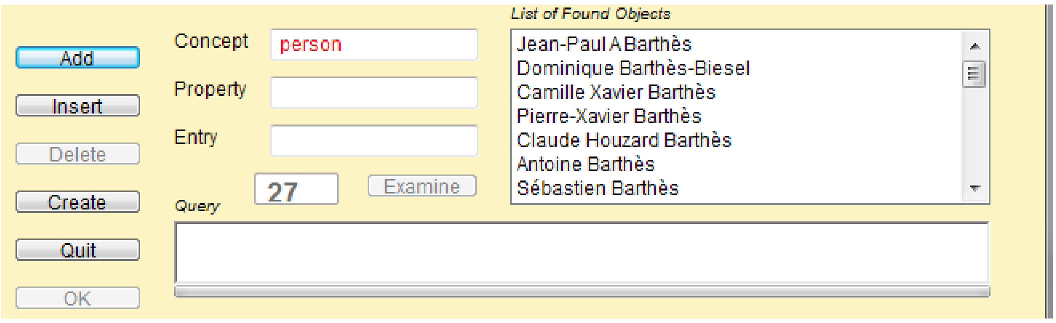

Figure 12.13: Displaying the objects of a given class (here PERSON)

Double-clicking an object of the list call the browser. It is a convenient way to have more information on the object.

Objects can also be located by using the query area, which works like in the window area. Executing the query is done by clicking the `enter` key. The number of candidates to be examined in the query is given in the counter. Viewing the objects is done by pushing the Examine button (Fig.12.14).

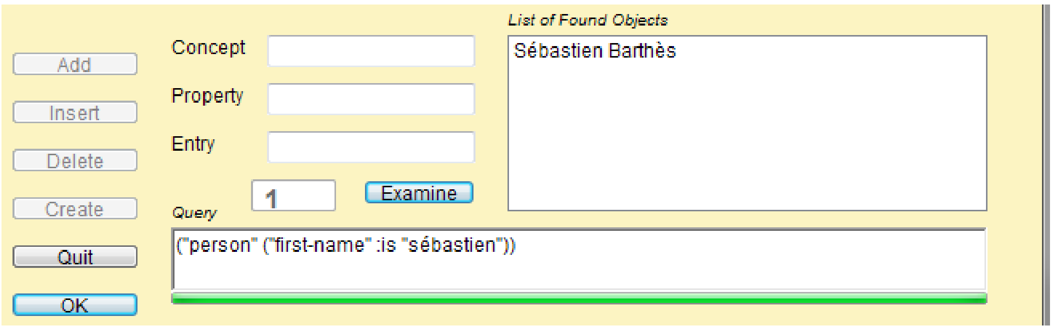

Figure 12.14: Result of executing a query for locating objects

Creating a New Object to Add

It is possible that the object that needs to be added does not exist. It is then possible to create it by creating a new instance or by cloning an existing object.

Creating a New Individual/ Instance

This can be done by typing a concept name into the Concept slot, and clicking the Create button. Then a message appears,

```
You want to create an instance of "PERSON"?
Yes, push OK; no, push CANCEL.
```

and if confirmed (OK button), then an empty object of the selected class is created to be filled with the necessary information. MOSS displays a new editor window. The editing process is recursive.

Cloning an Existing Object

If an object of the list of found objects is selected, then clicking the clone button brings up the message:

```
You want to clone "Sébastien Barthès"?
Yes, push OK; no, push CANCEL.
```

and if confirmed a new editor window with the cloned object appears. The cloned object can then be modified to create a new individual.

12.3.3 Editing a Relation

When the proper line of the left display window is selected and the Edit left button is clicked, then the linked objects appear in the top right window and the name of the relation is given to this window. The editing process proceeds as described in the previous paragraph.

For example clicking on the COUSIN line of the relations brings up the window shown Fig.12.15. The linked objects appear in the top right editing pane.

Selecting a cousin on the top right pane activates the editing buttons whose behavior is similar than in the case of attributes, except for the CANCEL and CREATE buttons already explained.

12.3.4 Deleting a Relation

To delete a relation, one must first select it on the left pane displaying relations and click the Delete button. No confirmation is asked.

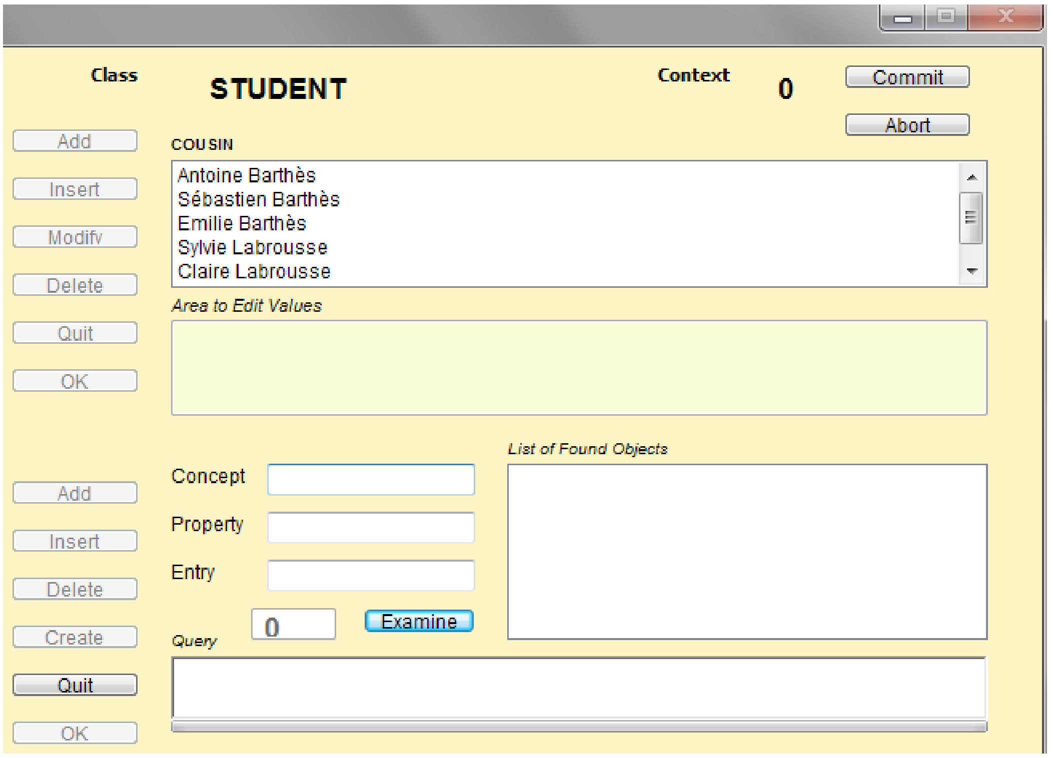

Figure 12.15: Editing the COUSIN relation

12.4 Following Inverse Links

When the inverse link area is activated it is possible to display the details of the objects related to the object being edited by double-clicking on it (Fig.12.16).

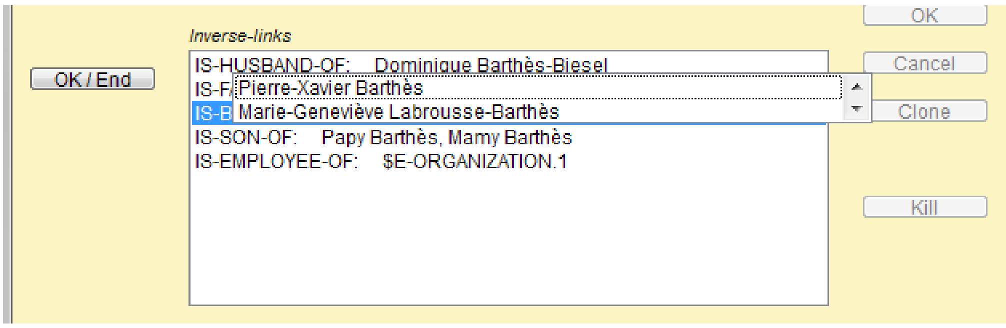

Figure 12.16: Viewing objects related by inverse relations

Clicking on one of the displayed objects calls a new editing window with the selected object to be edited.

Chapter 13

PDM4

Contents

13.1 Introduction

At the end of the 80s several attempts were done to combine the flexibility of an object-centered knowledge and data representation, with the modularity of object-oriented programming, and with the facility of permanent storage. Such a combination seemed necessary to cope with applications like CAD, office automation, or more generally AI based software. However, in some areas like conceptual design, or robotics, or simply prototyping applications, some difficulties were observed when using the available models. Indeed, in such domains, if objects were represented as instances, then they might have to change type and thus could not be attached to specific classes defined once and for all. On the other hand, environments using a prototype approach exhibited some difficulties for handling large number of objects because they suffered from organization problems. The PDM model was developed to combine both the necessary flexibility to accommodate the fluidity of changing situations, and the necessary structural strength needed to handle very large applications. The MOSS environment was developed to implement the model, and is still being developed today as needed. This document summarizes the main features of PDM version 4.

PDM version 4 adds several features: (i) multilingual facilities; (ii) multiclass belonging: (iii) virtual classes; (iv) synonyms; (v) generic properties.

Multilingual facilities are needed to develop multilingual ontologies (in particular in the European research programs [1]).

Multiclass belonging is also a result of ontology development, but curiously bring us back to the first implementations of the model when instances could be viewed according to different models. Virtual classes are classes that provide a special view on some instances. They themselves cannot have instances. They should not be confused with mixins.

Synonyms allow to use several expressions to designate the same object.

Since the first version of PDM, the vocabulary has been somewhat normalized. Since we are mostly interested in representing knowledge, we modified our own jargon to meet current use. Thus, the following changes have been introduced:

Previous name	**New name**
Entity	Concept
Terminal property	Attribute
Structural property	Relation
Instance	Individual

13.2 Some History

The PDM model is the result of a bottom up approach in which classes are abstracted from the existing representation of individual objects, and used as a soft specification for such objects. Individual objects do not belong necessarily to a class, and we use a mechanism similar to that proposed by Stein. Many object models have been proposed in the past both in artificial intelligence and in the database area, and more recently for developing ontologies in the context of the semantic web. In artificial intelligence models tend to act as prototypes and repositories for default values. Many models have been used

[1] We encountered the problem in the TerreGov IST project (2004-2007), for which we developed the SOL syntax.

for representing knowledge in languages like KRL [BW77], [GL80], UNITS [FK85], SRL [FWA86], AIRELLE [ANVPR88], SHIRKA [Rec85], CML [Sta86], and many others that have frame editor capacities.Within the database community attempts were undertaken in the eighties to cope with the need to support efficiently deductive databases and applications like CAD. Object models were proposed like [YOMF86], or O2 [LRV87]. In the database perspective object models tend to act as specifications for storing information.

In object-oriented programming languages the object representation is usually rather poor compared with what one can find in artificial intelligence, with however some interesting exceptions like LORE [BCP86] which used a set-theoretic object definition. Many papers, like one by Cointe [Coi87] tried to improve on the available object format.

Within our MOSS environment, objects are represented using an extended version of a model called PDM (Property Driven Model) which is a compromise between a frame model and a semantic data model (see Abrial [Abr74]). PDM like RLL is reflective (see Maes [Mae87] for a discussion about reflection). PDM has both specification (definitional) and prototyping capabilities together with some other interesting features described in the following paragraphs.

13.3 Basic Object Format

13.3.1 Properties

A PDM object is described by a set of properties, e.g. (see Fig.13.1)

```
name:    Einstein
first name:  Albert
birth country: Germany
specialty: Relativity
```

The two first properties have values (i.e. literals) directly attached to them and are called **attributes** (`att`), the two last ones are links to other objects (i.e. an object keyword for specialty and an object country for Germany) and hence are called **relations** (`rel`)[2]. The printed values Germany and relativity are actually a summary description of the corresponding objects.

Objects are instances of classes as shown Fig.13.1. In addition, but not shown in Fig.13.2, all structural properties (links between objects) are inverted, so that the object "`germany`" is linked to the object "`einstein`" by the property `is-country-of`.

The reader will have noticed that PDM does not use the classical 3-level frame-slot-facet format found usually in frame languages. Instead properties are themselves declared as objets and described using the same format, e.g.

name	
entity:	Concept
property-name:	NAME
is-attribute-of:	PERSON
method:	=if-added, =print-value, =entry

A relation is described in the same fashion:

specialty	
property-name:	SPECIALTY
is-relation-of:	Person
successor:	KEYWORD
inverse:	is-specialty-of
method:	=if-added

[2]In the OWL jargon attributes are called datatype properties and relations are called object properties.

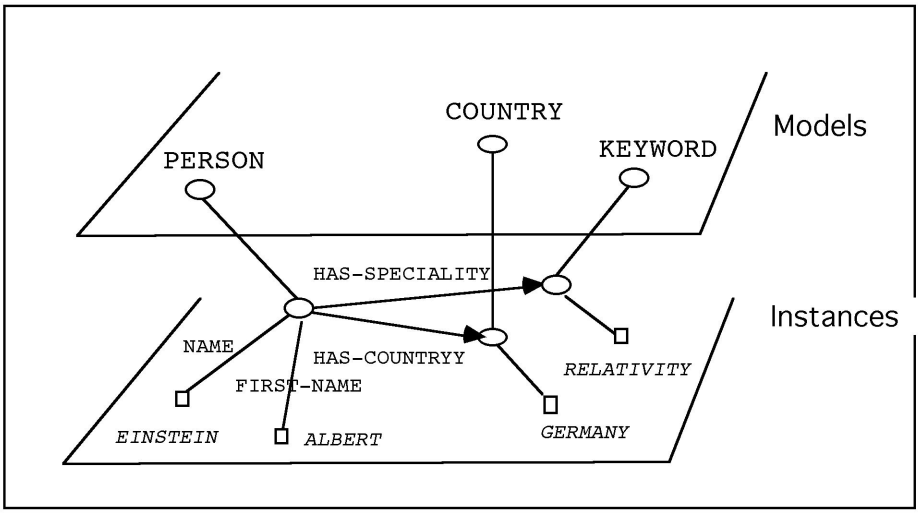

Figure 13.1: EINSTEIN Example

A long time ago, it was already mentioned by Cointe [Coi87] that such a reification of attributes would be very valuable in SMALLTALK.

13.3.2 (Soft) Classes

Classes are structural abstractions of instances. The class PERSON is described as follows:

Person	
entity-name:	Person
attribute:	has-name, has-first-name
relation:	has-specialty, has-birth-country
method:	=summary, =print-self,...

On the above examples it can be seen that information usually found in slots is factored in the attribute objects.

The class PERSON is represented Fig.13.2 in a layer, called models, containing class and property objects. The model layer is a specification layer describing the format (structure) of objects in the instance layer, as shown by the dashed lines on Fig.13.2.

In the PDM model an individual object (instance of a concept) need not obey the structural definition represented by the concept fully. Hence the term **Soft Class**.

13.3.3 Orphans

In addition to their properties all the previous objects has a special property called `HAS-TYPE` to indicate that they can be considered as instances of this class. E.g.

Einstein	
type:	Person
name:	Einstein
first name:	Albert
birth country:	Germany
specialty:	Relativity

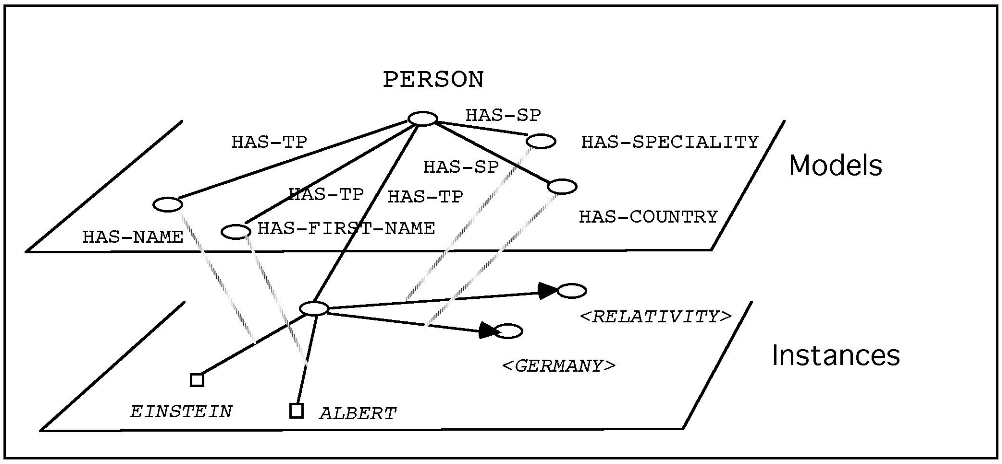

Figure 13.2: Details of the model specification layer

Some objects are special and do not belong to any class. They are called **orphans**. This happens for example when a robot encounters an object and does not know what kind of object it is, but acquires progressively more information. The object will remain an orphan until it has enough properties to be classified.

13.3.4 Defaults and Ideals

Properties (attributes and relations) can be used to specify default values. For example a typical PERSON may be called Smith, be a female and have age 40 and live in New York.

Attribute defaults are specific values (literals) and relation defaults are objects (instances or classes). Defaults can be attached to a special (virtual) instance of a class called an **ideal**. Alternatively, defaults can be attached to attributes directly (useful for orphans), e.g., `HAS-NAME` may record the default ?Smith.? Defaults attached to ideals have priority over defaults attached to properties.

13.3.5 Object Identifiers

Each object has a unique identifier, based on the radix of the class that defaults to the internal name of the class. E.g., `$E-PERSON.23`, 23 being the value of an instance counter at the time the instance was created. An object thus may be "empty" meaning that no attribute or relationship has any value, but nevertheless the object exists. Classes and properties are objects. The internal identifier for a class by default is built from its name being prefixed by `$E-`. The identifiers for attributes and relationships are built in the same fashion with respective prefixes of `$T-` and `$S-`.

Orphans have identifiers like `$0-12`, whose structure will be explained later.

Ideals have an identifier like an instance but numbered 0, i.e., `$E-PERSON.0`. They may or may not exist for a given class.

13.3.6 Inverse Relationships

In the PDM format all relationships (structure properties or links) are inverted. Thus, if a person has a brother, the relationship will be inverted to indicate that the other person is the brother of the first one. This allows to traverse the structure in any direction. MOSS maintains all inverse links automatically. Note that an inverse relationship is derived from the direct relationship and has a different semantics (more on this point later).

13.3.7 Inverse Attributes: Entry Points

Attributes can also be inverted and point onto a special object called an **entry point**, used as an index to locate the object. For example, the name ?Einstein? could be inverted and become an entry point. An entry point is an object whose identifier is the index, e.g. `EINSTEIN` in the previous case. The inverted name property (`IS-NAME-OF` internally `$T-NAME.OF`) points to the identifier of the Einstein object. Note that ?`Einstein`? could be the name of several persons or even the name of a company. Since our properties are multivalued, this is taken into account.

13.3.8 Demons

Inherited from artificial intelligence is the concept of **demon**. Demons are special functions (or methods) that are triggered automatically on special events. Several demons are very useful: `xi`, `if-added`, `if-removed`. Their functioning will be presented later.

13.3.9 Models and Meta-Model

Just as objects could be structurally abstracted to give birth to classes, classes and properties, that are objects, can be abstracted in turn as shown Fig.13.3.

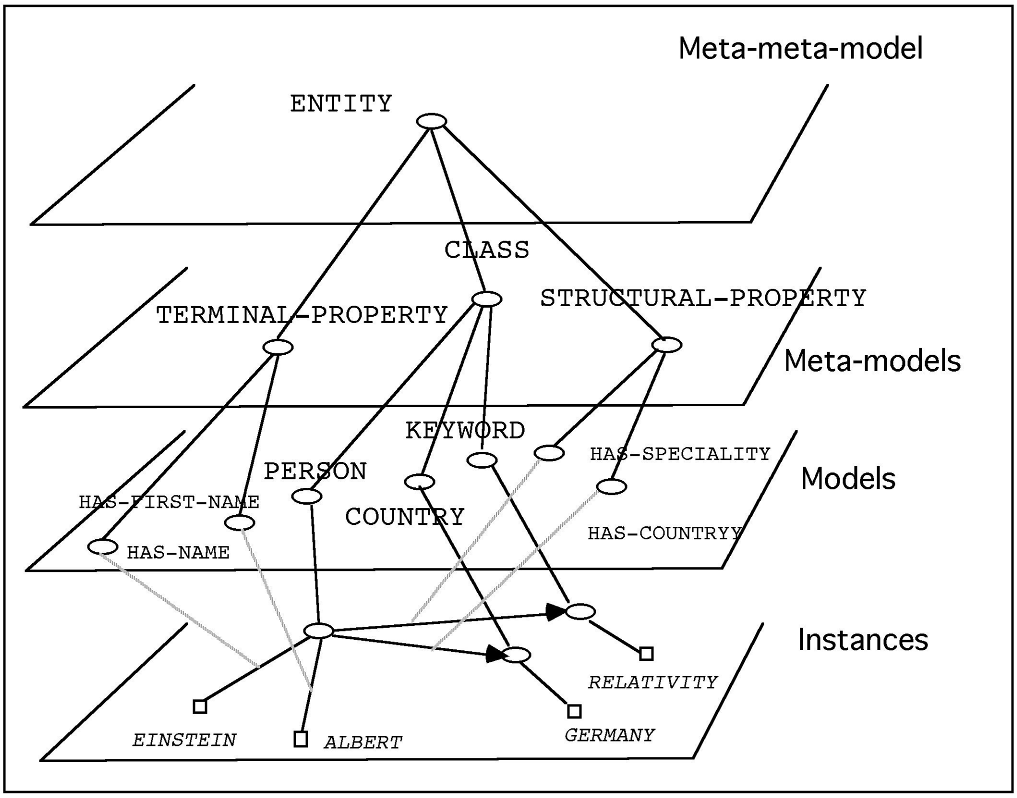

Figure 13.3: PDM specification hierarchy

Thus attributes (terminal properties in the figure), relations (structural properties) or concepts (classes in the figure) have models (superclasses), and, in turn those objects can be abstracted into a meta-meta-object that we call **entity**. To the question can the object entity be abstracted in turn, the answer is ?yes, of course.? However, entity happens to be its own model. Thus, *PDM is reflective* (it contains its own model) as many frame languages in artificial intelligence.

13.3.10 Value Inheritance, Prototyping

PDM offers the concept of **prototype**. Object A can be declared to be a prototype for object B. In that case, object A will provide all values that are not specified locally for B. Thus, if A and B are two objects and the shape of B is not known, then B will inherit the shape of A, a value. If B has a value, the value will shadow that of A.

Prototyping is useful when an object has no class but is known to look like or to behave as another object in the system.

13.3.11 Worlds, Views and Versions

PDM objects can be versioned, i.e., they can exist in different worlds or be looked or from different viewpoints. Versions in PDM are called **contexts** or **versions**. It is implemented as a world mechanism as for example in the former ART environment. It helps to solve problems like "Mary is 24 but Joe believes she is 22." PDM will create two worlds, one for the official numbers, and one for Joe.

All objects without exception can be versioned.

13.4 Behavior, the MOSS environment

The PDM model has been implemented in an environment called MOSS as an (extended) object-oriented language. Thus, MOSS implements message-passing, has methods and (extended) inheritance. The implementation was done in ANSI Common Lisp.

13.4.1 Message Passing

Objects can send or receive messages using the send function, e.g.

```
? (send cb '=print-self)
```

where the variable cb contains an object identifier (e.g. $E-PERSON.4). This yields:

```
----- $E-PERSON.4
 name:  Barthes
 first name:  Camille, Xavier
 sex: m
 sister: Peggy Barthes
-----
:done
```

A broadcast function can be used to send the same message to a number of objects. However, it does not execute in parallel.

13.4.2 Methods

Like in any OOL we use methods. PDM has three kinds of methods that are all first class objects: instance methods, own methods and universal methods.

Instance Methods

Instance methods are similar to methods in standard OOL, like Java. They are attached to classes and apply to instances of the class.

Own Methods

Own methods are attached directly to an object and can be used to specify the behavior of orphans, to express creation methods (like new) for classes, or to express exceptions.

Universal Methods

Universal methods are default methods that are not attached to any object in particular. They apply to all objects. Methods like `=print-self`, `=get-properties`, methods handling the object structure are universal methods.

13.4.3 Inheritance

PDM classes may have super-classes. However, PDM handles inheritance somewhat differently from standard OOL, due to the presence of prototypes (e.g., Fig.13.3) and of the different types of methods. When an object receives a message, it first checks if the corresponding method is an own-method, or if it can be inherited as an own method along an `IS-A` link. If not, then it asks its class for an instance method, either directly attached to its class or inherited via an IS-A link in the class specification layer. If not, then it checks if the method is not a universal method, in which case it is readily executed. If no method is found, then MOSS sends the object a message "`=unknown-method`" (which is a universal method).

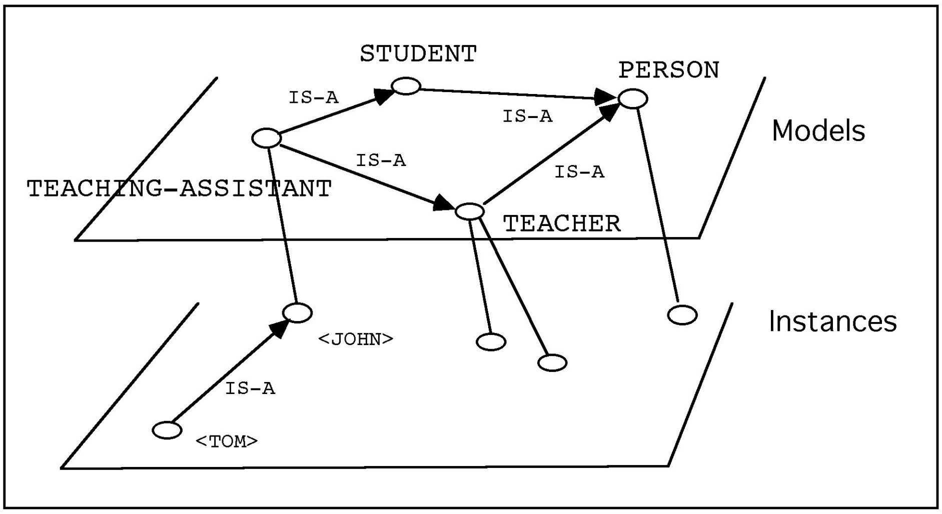

Figure 13.4: Inheritance hierarchy within a specification layer

Let us stress that a method is an object and as such can be versioned or can have methods.

13.4.4 MOSS Kernel

A number of classes and of methods have been defined to build the MOSS environment. Classes define the object structure, methods define the API. They are described in a separate document.

13.5 PDM as a Representation Language

A question that must be discussed is the following: PDM is an object model and MOSS is an Object-Oriented Language. But are the features presented in the previous sections good enough for a representation language, and in particular how is reasoning implemented?

A representation formalism should be discussed from two points of view: its expressiveness and its inference capabilities. We start with the second subject and will address the first one in Section 5.2.

13.5.1 Inference

PDM allows different types of reasoning. One is linked to the inheritance of values or methods. Another to the possibility of querying the system, a third one to the execution of methods, and a fourth one to the possibility of defining rules and executing them.

Defaults

Defaults are part of the PDM model and of the MOSS system. They are specified using the Ideal mechanism or at the property level. Noting special must be done.

Demons fall in the same category.

Queries

A query language, OQL, has been defined to extract subsets of objects from the system. Its syntax is fairly complex and detailed in a separate document.

Methods

As with any OOL methods can be written without any limit on what could be expressed.

Rules

As with some of the former frame languages (e.g., KEE or KC) one can define rules and executes them on the data, implementing standard expert systems, with various controls. One could also (but this is not implemented) define a method for doing PROLOG style resolution. This is used in some applications mostly for local bookkeeping rather than as an overall reasoning method.

13.5.2 Expressiveness

MOSS as it is, allows duplicating most of the mechanisms found in the various OOL (with the exception of multi-methods of CLOS), with additional features found previously in frame languages. This however does not confer a high representational power.

Extensional Representation

PDM objects represent actual objects whether concrete or abstract. The PDM representation is thus an extensional representation. A PDM class is not an intentional representation of the concept of PERSON for example, but a specification of the typical structure of instance objects. Thus, as it was defined intentional information must be expressed as methods to be applied (sometimes automatically, e.g. for demons) to the extensional representation of objects. Intentions buried within the code of methods are not available for reasoning.

Intentional Representation

We intend to add intentional capabilities to PDM by defining intentional objects, namely:

- An object representing the set of objects for a given class (this was already present in NETL);
- An object for representing some subset of objects of a class;

- An object representing a single unspecified object of a class.

This approach allows representing predicative formulae easily, but interferes with the previous mechanisms (e.g., the query mechanism), and thus needs more work

13.6 Database Features

Several extensions were done in the past for saving the objects onto disk and allowing multi-user access (LOB and VORAS environments). The persistency mechanism uses currently AllgroServe from Franz ACL environment. Please refer to Chapter **??**.

13.7 Multilinguism

Developing ontologies across different languages implies to be able to give different names in different languages, for concepts, but also for individuals. However, using the ontology is done within the context of a particular language. Thus, outputs should respect this approach.

13.7.1 Specifying Multilingual Names

Specifying names in different languages is done by using a new data type: a multilingual name; expressed as a list including word strings with language tags, e.g.

```
(:name :en "person" :fr "personne")
```

where the first tag(:name) is optional.
More formally:

```
<Multilingual-name> ::= ({:name} {<tag> <synonyms<}+)
<tag> ::= :en | :fr | :it | *
<synonyms> ::= "<text> {; <text>}* "
<text> ::= any string not containing " without escape \
```

When entering data, one can use the format with the provision that language tags are legal, i.e. defined at the system level.

13.7.2 Allowed Languages

Allowed languages are defined at the system level and have a corresponding language tag (e.g. `:EN` for English, `:FR` for French).

13.7.3 Current Language

When working in a specific environment, a current language must be specified. The current language filters name for display. The value of a multilingual name in a name whose tag it does not contain is "?".

13.7.4 Default Language

Default language is English. This is the language of the kernel.

13.7.5 Using Multilingual Names

Multilingual names may be used anywhere in inputs. Mixing languages is allowed.

13.8 Synonyms

13.8.1 Multilingual Name

Defining ontologies requires using synonyms. Synonyms are allowed when specifying multilingual names by using semi-columns in name strings, e.g.

```
(:en "person ; guy" :fr "personne ; quidam")
```

The number of synonyms is unlimited and spaces are not required before or after a semi-column.

13.8.2 Using Synonyms

Synonyms may be used for defining concepts, properties or individuals, e.g.

```
(defindividual "quidam" ("nom" "Albert"))
```

Synonyms can be used anywhere, and languages can be mixed in inputs, e.g.

```
(defindividual "guy" ("nom" "Joe"))
```

Each synonym generates an entry point (index object).

13.9 Generic Properties

Properties are created as independent objects, called **generic properties**. For example, a property ?name? will be created as a generic property independent from any class reference. If a class refers to that property, then a new property, attached to the specific class will be created, bearing the same name. Thus, a person will have a `"name"` corresponding internally to a `"person-name"`; a bookstore may have a `"name"` corresponding to a `"bookstore-name"` maybe with a different behavior, like different cardinality constraints. MOSS handles the differences automatically.

Generic relations are defined as relations from any object to any object (class ***any***), including classes or individuals.

13.10 One-Of

The ”one-of” construction is intended for constructing a class from a list of instances, meaning that the class comprises such instances and no others.

This does not fit the semantics of MOSS classes (concepts) since they are only meant to abstract the structure of a set of objects and not give the list of allowed objects, which represent a necessary and sufficient condition.

When applied to a class, the list of instances can be stored at the class level, associated with a `$TYPE.OF` property. The presence of such a property does not block make-instance functions. Indeed, the user in the MOSS philosophy is entitled to create objects that violate constraints, whenever necessary. MOSS issues a warning.

When applied to a property, then we consider two distinct cases: attribute and relation.

In the attribute case, a one-of indicates a series of possible choices for the value associated with the attribute. Thus, it can be associated with the property `$VAL`.

In the relation case it contains a list of possible successors of the specific property, i.e. the class of successors is an anonymous class made of the set of individuals specified by the `:one-of` option. If a `:to` property is also present then the individuals must be of the type specified by the `:to` option (range).

13.11 Restrictions

Additional restrictions may be imposed locally, i.e. to a property associated with a class. They are specified by the `:exist` and `:forall` options.

The `:exists` option indicates that among the values (for an attribute) or successors for a relation), one at least of the item must be taken from the :one-of list or be of the specified type when the `:one-of` option is not present. There must be at least one value.

The `:forall` option indicates that if there are items associated with a property, all of them must be taken from the one-of list or be of the specific type (when the one-of option is not present). There may be no value.

A special `:not` option indicates that values or objects should not belong to the one-of list.

Other operators are available for attribute values, namely, `:between` and `:outside`.

Cardinalities are treated as restrictions.

13.12 Multiple Types

An individual can belong to multiple classes, inheriting properties from each class. Like in CLOS, classes are ordered in such a way that properties inherited from classes in front of the list shadow properties inherited from other classes.

The internal implementation for selecting the object id selects the first class as the ”primary index” and the other classes as ”secondary indexes” or ”reference-ids,” introducing a level of indirection for addressing the object. Thus any object may be a reference-id meaning that it must be de-referenced to access its actual value.

Bibliography

[Abr74] Abrial. Data semantics. In J.W. Klimbie and K.L. Koffeman, editors, *Data Base Management*. North Holland, 1974.

[ANVPR88] Anne Adam-Nicolle, Bernard Victori, Christine Porquet, and Marinette Revenu. Airelle, rapport de présentation. Technical report, Université de Caen, Fevrier 1988.

[BCP86] Christophe Benoit, Yves Caseau, and Chantal Pherivong. The lore approach to object-oriented programming. Technical Report Memo C29.0, CGE, Laboratoires de Marcoussis, April 1986.

[BW77] Daniel G. Bobrow and Terry Winograd. An overview of krl, a knowledge representation language. *Cognitive Science*, 1(1):264–285, 1977.

[Coi87] Pierre Cointe. Metaclases are first class: the objvlisp model. *Special Issue of SIGPLAN Notices*, 22(12):441–451, 1987.

[FK85] R. Fikes and T. Kehler. The role of frame based representation in reasoning. *Communications of the ACM*, 28(9):904–920, 1985.

[FWA86] Mark S. Fox, Mark Wright, and David Adam. Experiences with srl: An analysis of a frame-based knowledge representation. In L Kerschberg, editor, *Expert database systems*, pages 161–172. The Benjamin/Cummings Publishing Company Inc, 1986.

[GL80] Russel Greiner and Douglas B. Lenat. A representation language language. In *AAAI-80*, pages 165–169, Stanford, 1980. Stanford University.

[LRV87] Christophe Lécluse, Philippe Richard, and Fernando Velez. O2, an object-oriented data model. Technical Report Altaïr, INRIA, 1987.

[Mae87] Patty Maes. Concepts and experiments in computational reflection. *Special Issue of SIGPLAN Notices*, 22(12):147–155, 1987.

[Rec85] François Rechenman. Shirka: Mécanismes d'inférence sur une base de connaissances centrée objets. In *5e Congrès AFCET sur la reconnaissance des formes et l'intelligence artificielle*, Grenoble, France, 1985.

[Sta86] Stanley. *CML: A Knowledge Representation Language with Applications to Requirements Modeling*. PhD thesis, University of Toronto, 1986.

[YOMF86] Xu Yu, Nobuo Ohbo, Takashi Masuda, and Yuzuku Fujiwara. Database support for solid modeling. *The Visual Computer*, 2(6):358–366, 1986.

Index