UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique

MOSS 7 - Kernel Methods

Jean-Paul Barthès

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

Email: barthes@utc.fr

N223
July 2008

Warning

The document presents a set of methods that can be used in a programming environment. With the exception of advanced format handling methods, they preserve the integrity of the objects according to the PDM4 model. They assume that all objects are in core or were brought into core before they are used. Default methods are provided for standard operations like creating entry points or checking cardinality. Such methods can be redefined by the user for any class.

The terms class, instance, terminal property, structural property have been dropped to be replaced by concept, individual, attribute and relation that will be used in the document. However, now and then class and instances are still used. The should be understood as synonyms for concept and individual.

The current research version of MOSS 7 runs in a MacIntosh Common Lisp environment (MCL 5.2 for OSX). It has been ported to Allegro Common Lisp (ACL 6.1 and 8.1 running under Windows XP).

Keywords

Object representation, object-oriented programming environment, kernel methods.

Revisions

Version	**Date**	**Author**	**Remarks**
1.0	Jul 06	Barthès	Version 1
1.1	Jul 08	Barthès	Upgrade to v7

MOSS documents

Related documents

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

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

Contents

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 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.

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 2 presents some methods progressively according to what the user intends to do, Section 3 presents them by nature (universal methods, methods attached to classes). Section **??** presents them alphabetically.

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 **??**.

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.

2.1.1 Creating an Attribute

There are several ways of creating attributes:

- Using defattribute macro
- Using the make-attribute function
- Sending a =new message to the object (class) representing an attribute (moss::$EPT)

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

```
? (defattribute COLOR)
$T-COLOR
? (m-make-attribute 'SHAPE)
$T-SHAPE
? (send 'moss::$EPT '=new 'date)
$T-DATE
```

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 two other methods are 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.

2.1.2 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
- Sending a =new message to ?*none*

For example we create 3 objects:

```
? (defobject ("COLOR" "Orange")(:var _o1))
$0-0
? (m-make-object '("SHAPE" "Square")'("color" "Red")'(:var _o2))
$0-1
? (send 'moss::*none* '=new '("COLOR" "Blue")'(:var _o3)(list "date" (get-universal-time)))
$0-2
? $0-2 ; internal format
((MOSS::$TYPE (0 MOSS::*NONE*)) (MOSS::$ID (0 $0-2)) ($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.

2.1.3 Viewing Objects

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

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

----- $0-2
 TYPE: *NONE*
 identifier: $0-2
 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.

2.1.4 Creating a Concept (Class)

Creating a concept can be done by:

- Using the defconcept macro
- Using the moss m-make-concept function
- Sending a =new message to the model of classes (moss::$ENT)

Examples

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

? (send _banana '=new '("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.

2.1.5 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
- Sending a =new message to the model of relations (moss::$EPS)
- 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 person 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
```

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.

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

2.2.2 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
```

2.2.3 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.

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.

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.

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.

2.5.1 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")))
```

2.5.2 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).

2.5.3 Adding Methods

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

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).

2.6.1 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
```

2.6.2 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.

2.6.3 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.

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
```

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.

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.

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.

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.

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."

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.

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.

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.

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.

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).

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.

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.

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.

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.

3.10 Methods Associated with Specific Objects

3.10.1 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.

3.10.2 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.

3.10.3 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 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 (universal)

=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 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 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 book-keeping 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 "Jérôme")(: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=0=0=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 (moss-system)

=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 *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 (universal)

=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 **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 **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 **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)

=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 (concept)

=instance-name **Instance method**

Returns the class name (as a list).

Example:

```
? (send _person '=instance-name)
("PERSON")
```

=instance-name (inverse link)

=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 (method)

=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'épie")
(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" "Gérard"))
"name: Albert, Gérard"
```

=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" "prénom")
 ("RELATION " "adresse domicile" "email" "page web" "epoux" "épouse" "mère")
 ("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 *&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)

=print-error **Universal method**

Deprecated.

=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 *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 (concept)

=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 (counter)

=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 (relation)

=summary **Instance method**

Returns the property name.

Example:

```
? (send _has-brother '=summary)
("BROTHER/UNIVERSAL-CLASS")
```

=summary **Instance method**

Example:

```
? (send moss::*moss-system* '=summary)
(MOSS)
```

=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.