UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique

MOSS 7 - Advanced Programming

Jean-Paul Barthès

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

Email: barthes@utc.fr

N221
July 2008

Warning

This document describes advanced features of the MOSS 7.xx programming environment for the experienced user. It should be used in conjunction with the MOSS 7 Kernel Methods Manual, the MOSS 7 Low-Level Functions Manual, and the MOSS 7 Primer. The document was upgraded from the previous MOSS 6 version.

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.

Revisions

Version	**Date**	**Author**	**Remarks**
0.1	Jul 06	Barthès	First version
1.0	Feb 07	Barthès	Adding multilingual names, synonyms, generic properties, multiple class belonging, allowing instances to change class
2.0	Jan 08	Barthès	Cleaning up
2.1	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

1 Introduction

The paper contains a description of some advanced features of the MOSS 7.xx programming interface. It is intended for the application designers who are concerned with efficiency, or who simply want to have a deeper insight on some of the issues. Some internal functions can be called directly instead of using kernel methods, which avoids going through the process of sending messages. When doing so, of course one is more efficient, however one looses the message tracing facilities.

The reader is referred to the N196 - PDM4 Memo for a description of the underlying model. In the following text we use the terms class and concept interchangeably. Likewise instance and individual.

2 Synonyms, Multilingual Names and References

2.1 Class Names, Property Names and Synonyms

Concept and function names are now allowed to be strings or multilingual names rather than symbols.

```
(defconcept HOUSE (:rel ADDRESS ADDRESS))
```

can be now written

```
(defconcept ?House? (:rel ?address? ?Address?))
```

The case for letters in the string is not important, however, it will be used for printing.

The consequences are the following:

- the name of the concept and properties does not depend on the package in which the functions are executed.
- it is now possible to add synonyms by separating them with a semi-column.

```
(defconcept ?House; building?
    (:rel ?address; living place? ?Address?))
```

2.2 Multilingual Names

Furthermore, it is possible to add multilingual names in the following way:

```
(defconcept (:name :en ?House; building? :fr ?maison?)
   (:rel (:name :en ?address; living place? :fr ?adresse?)
         ?Address?))
```

The leading marker `:name` is optional.

Thus, most of the names can be replaced by strings or multilingual names and are now called references. Most functions have been rewritten to accept references instead of symbols denoting classes or properties.

Note however that the string referring to properties does not contain the prefix ?HAS-?, MOSS being able to make the difference between classes and properties bearing the same name from the position of the reference.

3 Entry Points

Entry points are actually keys computed by the system to be used as indexes to access corresponding objects.

There is a default method =make-entry used when the `:entry` option is specified in one of the definition macros. When applied to a value (list) the method returns an atom, capitalized, each separate word being separated by a dash, e.g.

```
"Robin Hood"  yields the atom ROBIN-HOOD
```

Otherwise, a user-defined function can be specified with the :entry option, in which case the result must be a list of atoms, built in any way suitable to the user. For example the INTERMARC format for coding the French titles of books in catalog applications removes the leading article, then keeps the first 3 letters of the first word and then the first letter of the following three words, e.g.

```
"Le Petit Chaperon Rouge"  yields PETCR
```

Of course when one uses special functions for formatting entry points, one must apply them in exactly the same fashion to produce the key to access the corresponding objects.

Entry points need not be unique. Furthermore, entry points are instances (individuals) of an entry point concept. They usually do not have direct properties but rather contain a list of pointers to objects grouped by inverse attributes.

Since properties are multi-valued, an entry point is built for each value associated with the attribute. Nothing in practice prevents the =make-entry method to build several entry points associated with the same value. This can be interesting for range indexing, although the feature was never really tested.

4 Recovering an Internal Object Id

Internal object-ids are meaningless to the user. In addition, they can change during a prototyping phase when objects are created by downloading a text definition file. Thus, they cannot be used directly. However all messages must specify object-ids. When object-ids are contained in variables one uses the variable. When no variable was attached to the object (e.g. through the `:var` option in the defxx macros), then the object-ids must be retrieved from the system. The =id method attached to the class of entry points does the job.

=id *attribute class* **method**

=id returns the internal obj-id of an object when sent to the entry point corresponding to an attribute. The result is a list of all objects corresponding to the entry point.

Example

```
(send 'JONES '=id "name" "person")
```

returns the list of all persons (obj-ids) with name JONES.
This is not a query mechanism as can be found in the MOSS/QS subsystem.
See also the service function %extract for faster access.
It is also possible to use a query (access function) to recover the list of objects, e.g.

```
(access 'JONES)
```

or, using more control,

```
(access '("person" ("name" :is "Jones")))
```

5 Selectors and Filtering Functions

<must be rewritten>

6 Generic Properties

A MOSS property may be created in different ways: (i) directly, i.e. independently from any class; or (ii) within the context of a class (while defining a class).

In each case, a generic property is created first unless it already exists. Then, if the property is a class property, a local property is created, attached to the class.

Example:

```
? (defattribute "weight" (:entry) (:max 1))
$T-WEIGHT
```

The property weight is created as a generic property independently from any class. It can be used by any object in the system. However, a new property with the same name can be created and at the same time be attached to a class, e.g.

Example:

```
? (defattribute "weight" (:entry) (:class "person"))
$T-PERSON-WEIGHT
```

In that case the new property is created as sub-property of the generic property.

When creating a property attached to a class, the class must already exist, unless it is being defined, and a generic property is first created, then a specific sub-property attached to the class is being created. The name of the two properties are the same.

Example:

```
? (defconcept "person" (:att "name" (:entry) (:min 1)))
$E-PERSON
? (defindividual "person" ("name" "John")(:var _john))
$E-PERSON.2
? _john
$E-PERSON.2
? (send _john '=get "name")
("John")
? $E-PERSON.2
(($TYPE (0 $E-PERSON)) ($ID (0 $E-PERSON.2)) ($T-PERSON-NAME (0 "John")))
```

Warning: there is no check between the logical consistency of the options of a generic and a local property. I.e. a generic property could specify a maximal cardinality of 2, and a specific property with the same name a minimal cardinality of 3. The condition on the local property will shadow that of the global property. This point should probably be revised in the future. Example:

```
? (defattribute "color" (:max 1))
$T-COLOR
? (defattribute "color" (:class "person")(:min 2))
$T-PERSON-COLOR
? (send '$t-color '=get "maximal-cardinality")
```

```
(1)
? (send '$t-person-color '=get "minimal-cardinality")
(2)
```

When the generic property is first created, it takes the restrictions that are defined at that time. When creating local properties later with the same name, any restriction will apply only locally. I.e., the restrictions of the generic property will not be changed.

7 Multi-Class Belonging

It is now possible to create individuals belonging to several classes.

Example:

```
(defconcept "teacher" (:is-a "person"))
$E-TEACHER
? (defconcept "student" (:is-a "person"))
$E-STUDENT
? (defindividual ("student" "teacher") ("name" "albert"))
$E-STUDENT.1
? (send '$e-student '=get-instances)
($E-STUDENT.1)
? (send '$e-teacher '=get-instances)
($E-TEACHER.1)
? (send '$e-student.1 '=get "name")
("albert")
? (send '$e-teacher.1 '=get "name")
("albert")
```

Albert is a Teaching Assistant and a doctoral student, and thus can be considered as a teacher or as a student.

Internally there is a single object representing Albert. The second object is a reference to the first one (same technique as the Lisp ”invisible pointer”.

Warning: if we use the access function to access persons, then Albert will appear twice, once as a teacher and another time as a student:

```
? (access '("person"))
($E-PERSON.1 $E-PERSON.2 $E-PERSON.3 $E-TEACHER.1 $E-STUDENT.1)
```

This should be fixed in the future.

8 Internal MPS Functions

The message passing system is constituted by a set of internal functions that can be called directly. The functions rely on a format that implements objects as association lists. The MPS functions implement message passing and method inheritance. If they are modified, it is quite possible for the system to collapse.

8.1 Message Passing Primitive Functions

There are actually 4 basic functions: send, send-no-trace, broadcast, send-super.

send *object-id method-name &rest arguments* **function**

object-id must evaluate to the internal id of an object (otherwise see the remark below) *method-name* to the name of a method, *arguments* are the optional arguments required by the method.

Example:

```
(send _p1 '=get 'HAS-NAME)
(send _p1 '=add-tp-id '$T-PERSON-NAME "Albert")
```

assuming that _p1 contains the obj-id of the person we want to send the message to (e.g. $E-PERSON.34), and that $T-PERSON-NAME represents the obj-id of the property NAME.

The send function takes care of the tracing machinery and then calls another internal function send-no-trace. The latter function is necessary since the tracing mechanism, implemented by sending message, would enter an infinite loop if we had only one send function.

send-no-trace *object-id method-name &rest arguments* **function**

Mainly used by the system for avoiding either entering an infinite loop, or cluttering the screen with low level trace messages when tracing all traffic.
It is not advised to use this function, because when it is used the corresponding messages cannot be printed by the tracing mechanism. The overhead incurred by using the send function when tracing is off is not very significant.

broadcast *object-id-list method-name &rest arguments* **function**

The function takes the same arguments as send with the exception of the first one that must evaluate to a list of object-ids. The broadcast is not a parallel operation but more a serial broadcast to each object in turn (like a let* for example). It returns the list of all answers returned by each object to the message.

Example:

```
(broadcast (list _p1 _p2) '=get "first name")
returns
(("Jean") NIL)
```

sends to both _p1 and _p2 a message requesting the list of their first names. A nil value could come from an object with unknown first name, or no first name attribute.

send-super *method-name arg-list* **function**

The function is analogous to the SMALLTALK send-super, and is used within a given method code to acquire the method with the same name defined at a higher level.
This function should be used carefully.

8.2 Method Inheritance and Associated Functions

See the next paragraph on "The Inheritance Mechanism."

9 The Inheritance Mechanism

Inheritance uses two possible mechanisms:

1. a complex mechanisms allowing to modify inheritance at each stage by using inheritance methods,
2. a simplified traditional lexicographic mechanism.

The first mechanism is described in Sections 9.1 through 9.3. The second one is described in Section 9.4. The default mechanism is the simple one.

9.1 General Mechanism

When a given object receives a message it must locate the corresponding code. This is done using the following algorithm:

1. if the method name refers to an own method locally available, then the local own method is used;
2. if the own method is not available locally, then it is inherited from the ancestors of the receiving object. To do so MOSS sends to the receiving object a new message =inherit-own with argument the needed method name:
   (a) if a method =inherit-own exists locally then it is used to look for the method originally wanted;
   (b) otherwise the process recurses until eventually the default universal method =inherit-own is used, in which case inheritance is done lexicographically (depth first).
3. when no own method can be found, then one tries to obtain an instance method from the class of the object (classless objects are a special case). When the method is not readily available, then an inheritance mechanism similar to the previous one is used replacing =inherit-own with =inherit-instance.
4. When everything fails, one looks for a universal method. If there exists one, then it is used;

For classless objects, at step 2 one follows the own-method inheritance path until an object instance of a class is encountered. As soon as this happens the request is passed to the corresponding class for an instance method.

9.2 Method Inheritance

The inheritance mechanism is implemented by the get-method function. It calls three sub-functions, which have a very similar behavior as can be seen in the following descriptions of the algorithms we use.

moss::get-method *object-id method-name* **function**

The function returns the method name associated with the object whose id is passed as the second argument. It does so according to the inheritance procedure explained previously.

moss::get–instance-method *object-id method-name* **function**

The detailed algorithm is as follows:

1. If the object-id is nil, then we quit.

2. If the object already has the instance method we are looking for, cached locally, then we use it. If it has the indication that no such method is available, then we return nil.
3. If the method is available at the site as an instance-method of the object, then we reconstruct the LISP function from the method object building an internal unique function name, and we cache the result locally.
4. If everything failed so far and we have no more parents, then we record that the method is not available for this object, and we return nil.

   Here, we did not find the method, but we have some parents. We must explore the parents in the same order as the =inherit-instance has done, with the particularity that we are looking for an instance method and not for an own-method. Hence we are going to send the message =inherit-instance to the object (*self*). However, if the method we are looking for is =inherit-instance itself, then we must be careful no to enter an infinite loop.
5. If the method we are looking for is =inherit-instance, then since the default strategy for inheritance is depth first, we organize a depth first strategy to locate a possible special =inherit-instance. Hence we try to locate the special method depth first among the ancestors. When this fails the function returns nil.
6. If the method is not =inherit-instance, then we send ourselves the message =inherit-instance with the name of the method we are looking for. If we find it, then we cache it.
7. If everything failed, then we record that an instance method is not available for such an object.

moss::get–own-method *object-id method-name* **function**

The detailed algorithm is as follows:

1. If the object-id is nil, then we quit.
2. If the object already has the instance method we are looking for, cached locally, then we use it. If it has the indication that no such method is available, then we return nil.
3. If the method is available at the site as an own-method of the object, then we reconstruct the LISP function from the method object building an internal unique function name, and we cache the result locally.
4. If everything failed so far and we have no more parents, then we record that the method is not available for this object, and we return nil.

   Here, we did not find the method, but we have some parents, then we must explore the parents. Hence we are going to send the message =inherit-own-instance to the object (*self*). However, if the method we are looking for is =inherit-own-instance itself, then we must be careful not to enter an infinite loop.
5. If the method we are looking for is =inherit-own-method, then since the default strategy for inheritance is depth first, we organize a depth first strategy to locate a possible special =inherit-own-method. Hence we try to locate the special method depth first among the ancestors. When this fails the function returns nil.
6. If the method is not =inherit-own-method, then we send ourselves the message =inherit-own-method with the name of the method we are looking for. If we find it, then we cache it.
7. If everything failed, then we record that an own method is not available for the object.

moss::get-isa-instance-method *object-id method-name* **function**

The function is called when trying to inherit an own-method along the IS-A link did not work. The idea is thus to follow the is-a link until one find an object instance of a class, in which case one tries to inherit an instance method from the class as usual. If this does not work, then one follows the IS-A link one step further and tries again, until we end up with some object having no parents, in which case we quit. When coming down the exploratory tries, we mark the passed classes with either the method if we found one, or with the indication that there is none if we found none.

It is important that we go up the IS-A inheritance link with the =inherit-own method to follow the inheritance path in the same fashion as when we looked for an own-method.

The detailed algorithm is as follows:

1. If the object-id is nil, then we quit.
2. If the object already has the instance method we are looking for, cached locally, then we use it. If it has the indication that no such method is available, then we return nil.
3. If the object is an instance of some class, then we call the get-instance-method on its class. Note that we do not allow an object to belong to multiple classes, otherwise this part should be changed.
4. If everything failed so far and we have no more parents, then we record that the method is not available for this object, and we return nil.

   Here, we did not find the method, but we have some parents. We must explore the parents in the same order as the =inherit-own has done, with the particularity that we are looking for an instance method and not for an own-method. Hence we are going to send the message =inherit-isa-instance to the object (*self*). However, if the method we are looking for is =inherit-isa-instance itself, then we must be careful no to enter an infinite loop.
5. If the method we are looking for is =inherit-isa-instance, then since the default strategy for inheritance is depth first, we organize a depth first strategy to locate a possible special =inherit-isa-instance. Hence we try to locate the special method depth first among the ancestors. When this fails the function returns nil.
6. If the method is not =inherit-isa-instance, then we send ourselves the message =inherit-isa-instance with the name of the method we are looking for. If we find it, then we cache it.
7. If everything failed, then we record that an instance method is not available for such an object.

The trick is to have an =inherit-isa-instance method that works exactly as the =inherit-own.

moss::get-universal-method *object-id method-name* **function**

The function is called when the other inheritance functions failed. MOSS then tries to find a universal method corresponding to the method name. If there exists one, then it returns it otherwise it return a null method.

moss::find-own-method *object-id method-name* **function**

The function tries to locate a method locally, i.e. within the list of methods attached to the object. If none is found then returns nil.

moss::find-instance-method *object-id method-name* **function**

The function tries to locate a method locally, i.e. within the list of methods attached to the object. If none is found then returns nil.

9.3 Local Service Functions

moss::parents *object-id* **function**

The function returns the list of all objects parents of the current object. If none, then it returns an empty list.

moss::ancestors *object-id* **function**

Same as parents but returns the transitive closure of the parent objects.

moss:: gamma *object-id* **function**

The function is used to compute a transitive closure along any of the structural properties.

9.4 Simplified Inheritance

The fancy inheritance mechanism described in the above paragraphs is not always very useful. Thus, in simple situations, the user can turn it off in favor of a more traditional inheritance scheme, i.e. a lexicographic one (inheritance depth first).

The inheritance mode is controlled through the moss::*lexicographic-inheritance* global variable. When MOSS is booted in a standard fashion the inheritance mode is the simple lexicographic one.

10 Methods Caching

10.1 Rationale

Some level of optimization was introduced to avoid inheritance look up at each cycle. Methods are cached at the class level for instance methods and at the object level for own methods onto the p-list of the corresponding obj-ids. Failures are also recorded.

When methods are cached, then execution is slower at the beginning of a work session, but much faster after that.

10.2 The moss::*cache-methods* Global Variable

The caching mechanism is difficult to use when developing a new application, because methods have to be modified constantly. Hence a special global variable *cache-methods* can disable the process of caching them locally. This leads to a slower execution, since methods are inherited again and again for the same objects, but it is much safer and prevents obscure errors.

By default caching is disabled. To turn it on, it is enough to set the global variable moss::*cache-methods* to T.

11 Internal Service Functions

Many functions are provided for increased efficiency. They deal explicitly with versions and are rather difficult to use. The reader should refer to the corresponding Manual: MOSS 7: Low Level Functions. The following functions are given as examples.

moss::%extract *att class &key context filter class-ref function allow-multiple-values* **function**

Option (:filter <function>) When the filter option is specified then function is applied to the whole list of objects prior to returning them. If the result returns more than one value and the option allow-multiple-values is not set to T, an error is signaled.

Example

```
(moss::%extract 'SMITH 'HAS-NAME 'PERSON :context 25 :allow-multiple-values t)
```

will try to extract the persons, students,..., named Smith in context (version) 25.

moss::%extract-from-id *entry att-id class-id &optional context* **function**

Returns the list of all objects corresponding to entry-point *entry*, to the attribute *att-id* and to the class *class-Id*, in the specific *context*. Specifying the context is optional; default is current context.

Objects are returned either if they are instances of the class or of one of the subclasses, regardless of the system to which classes belong (i.e. a sub-class may belong to a different system than its superclass).

The function is in fact a first a first attempt to locate objects, that will be further filtered by the %extract function.

Example

```
(moss::%extract-from-id 'ATTRIBUTE 'moss::$ENAM 'moss::$ENT 0)
```

returns

```
(MOSS::$EPT)
```

moss::%make-ep *entry tp-id obj-id &optional context* **function**

Add a new object to an entry point if it exists otherwise create it and record the creation in the MOSS system.

Returns the value of the entry point (useful when using step).

Example

```
(moss::%make-ep 'DE.LA.PORTE '$T-PERSON-NAME '$E-PERSON.356)
```

12 The Error Mechanism

In case of a failure (no method corresponding to the message could be located) MOSS sends the special message =unknown-method to the receiving object. The whole process starts again.

13 Trace and Stepping Mechanisms

13.1 Global Control Variables

moss::*trace-flag* controls the printing of the messages sent and of the values that are returned.

moss::*trace-level* sets the column indentation for starting printing the trace.

13.2 Trace functions

Tracing can be done in different ways: global tracing of all exchanged messages, tracing of a particular method, or monitoring the message traffic for a particular object.

trace-message **function**
allows monitoring message traffic between objects.

untrace-message **function**
reverts to no trace situation.

untrace-all **function**
applies to all trace modes.

trace-method *method-name* **function**
prints messages using a particular method.

untrace-method *method-name* **function**
suppresses monitoring messages for the method method-name.

trace-object *object-id* **function**
prints the message traffic related to a particular object.

untrace-object *object-id* **function**
reverts to no monitoring.

14 The Meta-Model Level (Kernel)

Fig.1 pictures the bottom up approach and Fig.2 the abstraction/specification relationships between the instances and the classes (models or structural specifications).

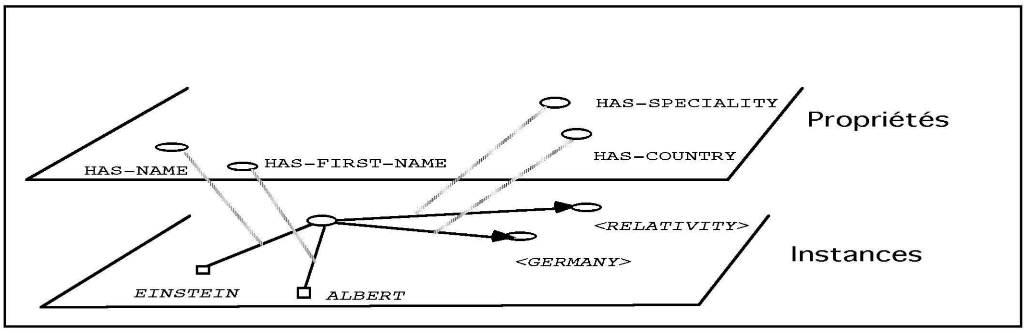

Figure 1: A class (concept) can be defined by abstracting the structure of the objects HAS-TP indicates an attribute (immediate value) HAS-SP indicates a relation (link onto another object)

Fig.3 shows for the same model the (multiple) inheritance structure. Inheritance is a class/sub-class relationship among objects. At instance level such links become prototyping links, in particular for orphans. Such classless objects are interesting for design applications or for recognition (like in robotics).

The main point here is a clear distinction between specification and prototyping, which was already mentioned by Stein 87.

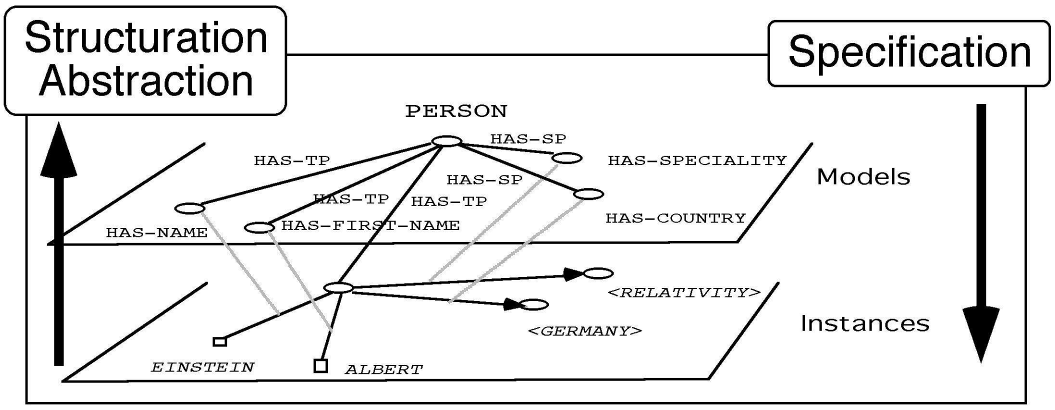

Figure 2: Objects (entities) are defined as soon as properties (quiddities) are defined. Properties specify the object structure.

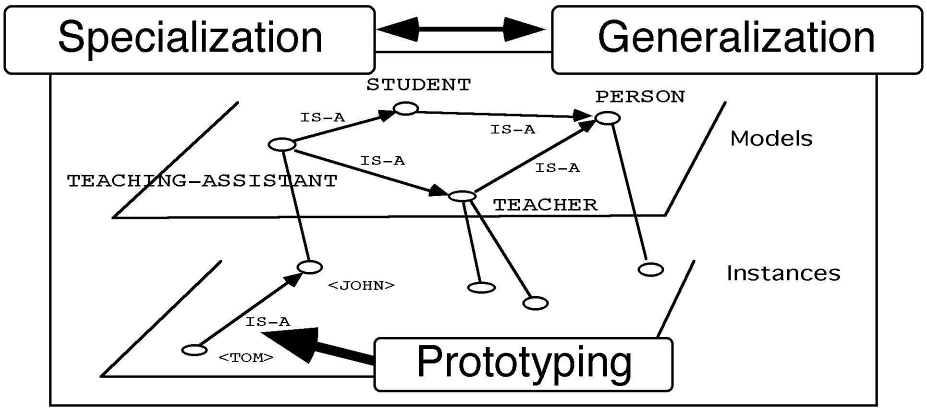

Figure 3: Specialization/generalization relationships are represented in the upper layer - In the instance layer the same is-a link is used to implement a prototyping relationship in particular for orphans.

Once classes have been defined, the abstraction process can be repeated at their level. Classes may be abstracted, i.e. their structure can be specified by meta-classes. In turn meta-classes can be abstracted, and so on until one finds an object which specifies its own structure (named here meta-meta-model). Fig.4 displays a 4-level hierarchy showing the reflectivity of the PDM model. Reflectivity is a powerful concept, since it allows to modify dynamically the structure of the model itself; however it is difficult to handle (see also Maes 87). Reflectivity can be the possibility of describing its own structuration like here. In KSL (Ibrahim & Cummins 88) the system specifies also the programming syntax of the language.

Reflectivity is important because it allows modeling the structure of the used representation. This is good for automatically providing **explanations**, and also for **learning**.

The two higher levels of the model must be hand-crafted, they constitute the kernel of the representation model.

15 Semantics of the Inverse Links

Inverse links do not bear any semantics. They are purely internal links to allow navigation along the arcs in both directions. This is used by the query system to optimize accesses. It is used of course by

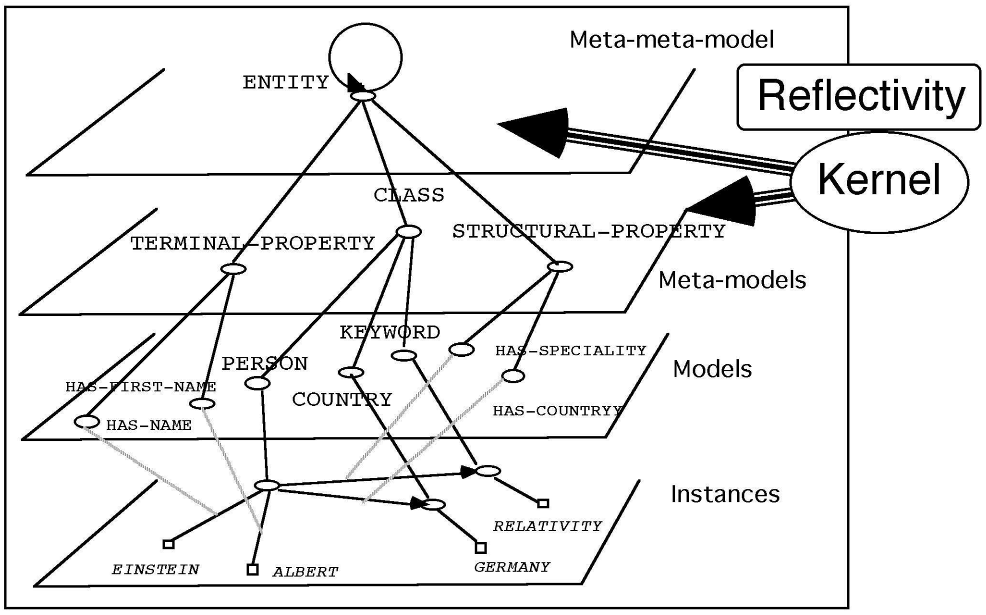

Figure 4: A (very simplified) representation of the various layers of the PDM4 model

the browser. Thus no integrity constraint are attached to inverse links, e.g. there cannot be things like a minimal or maximal cardinality. The facility of giving specific names to inverse links is intended to make rephrasing of queries more legible.

It is important to notice that if a person is linked to another person by the property HAS-FATHER, then the inverse link is by default IS-FATHER-OF, but it does not appear in the printing list of the second object. If one wants to state that the second object is linked to the first by a property HAS-CHILD, then one must do it explicitly. This may look somewhat heavy, but the benefits are quite substantial in the query system. I tried once to give a meaning to inverse properties, thus leading to integrity constraints attached to the inverse links. This leads to severe locking problems in the concurrency control mechanism.

16 Default Values

Semantic representations are expected to accommodate default values. A natural way of storing them is at the property level. However, with PDM3, because properties are shared among objects, there can be a problem. Consider for example the property name that has the same meaning and function whether it is applied to a person, a dog, or a company. If we want to store a default name at the property level then there may be a conflict since we are in essence specializing the shared property to one of the objects or concepts.

To cope with this problem, PDM4 uses the concept of **ideal**[1]. Every concept has an associated virtual individual instance, called its ideal, which bears all the defaults associated with the class. Thus, the ideal can be viewed as a typical individual instance of the concept. Currently, the defaults are limited to attributes. Whenever an object is built using the defindividual macro, then an ideal is created. It bears the sequence number 0. I.e., the ideal for a person will be $E-PERSON.0 if the radix for person is $E-PERSON. Actual instances will have internal ids starting at 1, i.e., $E-PERSON.1, $E-PERSON.2, etc. The ideal will behave as any individual with respect to MOSS methods. However it is not advised to temper directly with its content.

[1]The name refers to the notion of idea used by Plato.

16.1 Setting Defaults in Relation to Concepts

Setting the ideal through defconcept

Whenever a `:default` option is used for an attribute while defining a class, then the default value associated with the option is recorded in the corresponding ideal. E.g.

```
(defconcept person
  (:att AGE (:unique)(:default 30))
  (:att NAME (:min 1)(:max 3)(:default "Dupond"))
  (:rel BROTHER PERSON))
```

The subsequent ideal will have ”Dupond” as name and 30 as its age.

Setting the ideal through defattribute

The same result is achieved when defining the attribute separately with the `:class` option.

```
(defattribute address
  (:class person)
  (:default "Unknown"))
```

In this case however, the `:class` option must precede the default option in the list of options.

16.2 Setting Defaults Independently from Classes

It is possible to associate a default value directly to an attribute when defining it. Then the default will be associated directly with the attribute, independently of any class. It will stand for a typical value of this attribute. Assuming that we live in a *dark world* the default value for the concept of color could be defined as black, and stated as follows

```
(defattribute color
  (:default "Black"))
```

16.3 Use of Defaults

When the user ask for the value of an attribute of a given object, then the =get or =get-id method uses the following algorithm:

1. if the attribute has a local value, then return it;
2. otherwise, climb up the chain of prototypes (objects linked by an is-a property). If some object in the chain has a local value, then return this value;
3. otherwise, if the original object is an individual instance of a concept (or of several concepts), then look at the corresponding ideals for a default value. If one is found, then return it;
4. otherwise, look at the object defining the attribute. If a general default is present then return it;
5. otherwise, return nil.

16.4 Printing the Ideal (Typical Instance)

It is possible to print the ideal of a concept to see al the default values by using the =print-typical-instance method, e.g.,

```
(send _person ’=print-typical-instance)
```

17 The *any* and *none* Pseudo Classes

A `*none*` marker in the $TYPE field of an object indicates an orphan?classless object. On the contrary `*any*` indicates that the object may belong to any class defined so far, thus standing for a virtual root of all IS-A lattices.

any and *none* have been declared as pseudo-classes and implemented as such. They are both of type ENTITY.

Having explicit classes could bring some simplifications in the global functioning. Indeed, orphans can be declared to be instances of the *none* class and attached to it. They would not need a special mechanism for their creation. They could also be printed easily as instances of that class. On the other hand universal methods could be attached to the *any* class as instance methods since it represents any object in the system.

Currently, what can be gained and what problems will arise from doing that is not clear. Some experience must be gained before deciding whether it is a good idea to do so.

18 Adding Extra Properties

A problem arises when we want to add a property to an object that is an instance of a class or several classes and the property does not belong to such classes, i.e. the object is an exception with regard to its classes. The philosophy of PDM/MOSS is to allow adding such properties.

When adding an extra property to an object, the only possibility short of adding the property at the class level is to use the generic property, since a local version of the property does not exist for the instance. When doing so, MOSS was previously issuing a warning:

```
? (send _e1 '=add-attribute-values "name" '("message 1"))
; Warning: Attribute HAS-NAME cannot be found when processing object $E-EMAIL-TEST.1
; While executing: MOSS::*0=ADD-ATTRIBUTE-VALUES
NIL
```

The new version adds the property and issuing the following warning:

```
? (send _e1 '=add-attribute-values-1 "name" '("message 1"))
; Warning: Attribute HAS-NAME does not belong to the object ($E-EMAIL-TEST.1) class(es). We ad
; While executing: MOSS::*0=ADD-ATTRIBUTE-VALUES
:done
```

19 Individuals Changing Concept

From version 6.1 a new method how been added to let an individual instance change concept. However this raises some issues.

The case of individuals changing concept is encountered when new information has been found about an object represented by an individual, and one wants to refine its position. E.g., we just learnt that a person is following courses, and we want to associate the individual to the concept of STUDENT rather than PERSON. When the new target concept is a sub-concept of the original concept, there is no difficulty in doing it. However, if the concept is completely different, then some of the properties of the object may not appear as properties of the new concept. But since they are properties of the object, they should be kept. The question is: should we keep such additional properties as original local properties, or should we move the property to its generic value?

Assume that we want to move a student individual to become a teacher individual. What happens to the property HAS-COURSES that was declared locally to student class? The student individual will become a teacher individual. Should we keep the local property (internally $T-STUDENT-COURSES) or use the generic property ($T-COURSES)?

When adding an extra property we use the generic property. Doing so, we could kill all the local features that were attached to the local property.

Warning: if an object has extra properties and one adds the property as a local properties of one of the classes of the object, the values that were associated with the extra property will not be reachable any longer.

Warning: the current mechanism is not compatible with multiple concept belonging.

20 More on Multilingual Names

Multilingual names are tricky and the corresponding operational semantics must be defined carefully.

20.1 Purpose

The purpose of multilingual names is to be able to designate an object by several names in different languages, and to be able to select the corresponding name or documentation according to the active language (specified by the `*language*` global variable).

20.2 Possible Implementations

Several ways are possible for implementing multilingual approaches, among which: (i) using versions; or (ii) using a new datatype.

20.2.1 Version Approach

In the version approach we associate a language with a particular version. Thus, different languages will be associated with different parallel versions. We assume that the root version uses a default language, say English.

Versions are ruled by a configuration tree, in which objects *inherit* values from above. Thus, values can be shadowed by simply specifying a set of values for a given property. For example if we specify in version 1 that the property name for the object $PNAM is "Nom de propriété", then the version 0 value, say "Property name" is shadowed. However, the corresponding entry point PROPERTY-NAME is still reachable and points towards the right object.

Internal representation of the corresponding object:

```
(?($PNAM (0 ?Property name?)(1 ?Nom de propriété?))?)
```

Getting the property name when in version 1 will simply return the French text.

The main advantage of this approach is that nothing needs to be changed, using a particular language amounts to select the corresponding version, namely use an in-version wrapping macro.

The main drawback of this approach is that it generates a serious conflict with the other potential uses of the versioning mechanism (temporal changes, different hypothesis). Indeed, the language issue is orthogonal to the use of versions.

20.2.2 Datatype Approach

The datatype approach consists in creating a new datatype whenever a multilingual text or name is needed. The value is then a vector in which entries correspond to different languages, e.g.:

```
(?Property name? ?Nom de propriété?)
```

A better way consists in using language tags:

```
(:en ?Property name? :fr ?Nom de propriété)
```

Internally this can be represented in many different ways including strings like:

```
?Property name@en; Nom de propriété@fr?
```

The exact form of the internal representation does not matter as long as it allows to express text freely and functions are available to manipulate the different linguistic components.

The main advantage of this approach is that it is independent from the version mechanism.

The main drawback of the approach is that it is necessary to introduce functions and predicates associated with the new datatype.

The approach offers another interesting possibility, namely that of having all the languages available in the same environment, allowing to use any entry point to designate an object. This would not be possible in the version approach where one is restricted to the use of languages contained in the same branch of the configuration tree.

20.2.3 MOSS Approach

Within MOSS, I chose to extend the datatype approach with the possibility of including synonyms in the same language. The internal format would be:

```
({:name} :en ?person; individual? :fr ?personne ; individu?)
```

20.3 MOSS Operational Semantics

Selecting the datatype approach has several consequences. In particular, we must consider two cases: (i) creation time; and (ii) run time.

20.3.1 Creation Time

Creation time or loading time is the time when the internal object structures are created including, objects, entry-points, methods, functions, etc. The structures will then be used later at runtime.

At creation time one must be careful to build the right structures. Two situatios might occur: (i) we select a specific language; or (ii) we take all languages.

In the first case we can strip all multilingual information and drop the language tags. This is particularly interesting in the input files where the syntax may be simplified, the input language being specified by the value of the `*language*` global variable (e.g. `:en` or `:fr` or `:it`). Furthermore, there is a global variable named `*language-tags*` containing the set of legal languages, which allows us to introduce a specific short hand notation at creation time as follows:

```
({:name} :fr ?Londres? :* ?London?)
```

meaning that the name in French is Londres, and for all other legal languages is London. MOSS is then allowed to produce entry points corresponding to the different languages, using the necessary =make-entry methods.

If we select a specific language when loading an ontology, then the rationale is to keep only concepts and properties that have a name in the specified language. This can be done by running a trimming process that eliminates all concepts or properties that are not defined in the specified language and reducing the multilingual names to the synonyms in the specified language[2]. Globally, we obtain a

[2]The corresponding process can be quite complex.

projection of the ontology in the world corresponding to the specified language. This approach is not currently implemented in the MOSS/OMAS environment.

If we specify that we want all languages (`*language*` is `:all`), then we must produce all the concepts, properties, individuals, entry points in all the languages specified locally. In that case, the property names linking the objects in the input data set can be in any language.

20.3.2 Run Time

We can encounter two different situations at runtime: (i) we are in a monolingual environment; (ii) we use a specific language, but the objects were created in a multilingual environment.

In the first case (monolingual environment), we have no more multilingual data. Thus, the `*language*` global variable is useless.

In the second case, we must extract for each piece of information the right label or text in the specified language. A number of multilingual primitives allow doing that.

20.4 Internal Coding of Names

Multilingual names pose coding problems that can only be resolved by using UNICODE (in practice UTF-8). This leads to some problems in the sense that UTF-8 is not supported by all Lisp environments (specifically MCL[3]).

[3]MCL 5.2 supports UTF-8, however one must be careful when rewriting files.