UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique

Property Driven Model (PDM4)

Jean-Paul Barthès

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

Email: barthes@utc.fr

N196
January 2008

Warning

This document describes PDM (Property Driven Model) as implemented by MOSS. It describes in particular its new multilingual capabilities, and the possibility for instances to be members of different classes.

Keywords

Object representation, knowledge representation, object-oriented programming environment, frames.

Revisions

Version	Date	Author	Remarks
1.0	Jun 90	Barthès	Initial issue
2.0	Oct 03	Barthès	Revised version PDM3
3.0	Dec 05	Barthès	PDM4
3.1	May 08	Barthès	PDM4 Upgrades

MOSS documents

Related documents

- UTC/GI/DI/N196 - PDM4
- UTC/GI/DI/N197 - MOSS 6 : Dialogs
- UTC/GI/DI/N200 - MOSS 6 : Primer
- UTC/GI/DI/N201 - MOSS 6 : Programming Environment Advanced features
- UTC/GI/DI/N202 - MOSS 6 : Kernel
- UTC/GI/DI/N203 - MOSS 6 : Low Level Functions
- UTC/GI/DI/N204 - MOSS 6 : Syntax
- UTC/GI/DI/N205 - MOSS 6 : Query System
- UTC/GI/DI/N206 - MOSS 6 : User Interface (Macintosh Environment)
- UTC/GI/DI/N207 - MOSS 6 : User Interface (Windows Environment)
- UTC/GI/DI/N209 - MOSS 6 : Dialogs v0.2

Readers unfamiliar with MOSS should read first N196 and N200 (Primer), then N204 (Syntax), N206 or N207 (User Interface). N202 (Kernel) gives a list of available methods of general use when programming. N205 (Query) presents the query system and gives its syntax. For advanced programming N203 (Low level Functions) describes some useful MOSS functions. N209 (Dialogs) describes the natural language dialog mechanism.

Contents

1 Introduction **7**

2 Some History **7**

3 Basic Object Format **8**
3.1 Properties . . . 8
3.2 (Soft) Classes . . . 9
3.3 Orphans . . . 9
3.4 Defaults and Ideals . . . 10
3.5 Object Identifiers . . . 10
3.6 Inverse Relationships . . . 10
3.7 Inverse Attributes: Entry Points . . . 10
3.8 Demons . . . 10
3.9 Models and Meta-Model . . . 10
3.10 Value Inheritance, Prototyping . . . 11
3.11 Worlds, Views and Versions . . . 11

4 Behavior, the MOSS environment **11**
4.1 Message Passing . . . 12
4.2 Methods . . . 12
4.2.1 Instance Methods . . . 12
4.2.2 Own Methods . . . 12
4.2.3 Universal Methods . . . 12
4.3 Inheritance . . . 12
4.4 MOSS Kernel . . . 12

5 PDM as a Representation Language **13**
5.1 Inference . . . 13
5.1.1 Defaults . . . 13
5.1.2 Queries . . . 13
5.1.3 Methods . . . 13
5.1.4 Rules . . . 13
5.2 Expressiveness . . . 14
5.2.1 Extensional Representation . . . 14
5.2.2 Intentional Representation . . . 14

6 Database Features **14**

7 Multilinguism **14**
7.1 Specifying Multilingual Names . . . 14
7.2 Allowed Languages . . . 15
7.3 Current Language . . . 15
7.4 Default Language . . . 15
7.5 Using Multilingual Names . . . 15

8 Synonyms **15**
8.1 Multilingual Name . . . 15
8.2 Using Synonyms . . . 15

9 Generic Properties **15**

10 One-Of **16**

11 Restrictions **16**

12 Multiple Types **16**

1 Introduction

At the end of the 80s several attempts were done to combine the flexibility of an object-centered knowledge and data representation, with the modularity of object-oriented programming, and with the facility of permanent storage. Such a combination seemed necessary to cope with applications like CAD, office automation, or more generally AI based software. However, in some areas like conceptual design, or robotics, or simply prototyping applications, some difficulties were observed when using the available models. Indeed, in such domains, if objects were represented as instances, then they might have to change type and thus could not be attached to specific classes defined once and for all. On the other hand, environments using a prototype approach exhibited some difficulties for handling large number of objects because they suffered from organization problems. The PDM model was developed to combine both the necessary flexibility to accommodate the fluidity of changing situations, and the necessary structural strength needed to handle very large applications. The MOSS environment was developed to implement the model, and is still being developed today as needed. This document summarizes the main features of PDM version 4.

PDM version 4 adds several features: (i) multilingual facilities; (ii) multiclass belonging: (iii) virtual classes; (iv) synonyms; (v) generic properties.

Multilingual facilities are needed to develop multilingual ontologies (in particular in the European research programs [1]).

Multiclass belonging is also a result of ontology development, but curiously bring us back to the first implementations of the model when instances could be viewed according to different models. Virtual classes are classes that provide a special view on some instances. They themselves cannot have instances. They should not be confused with mixins.

Synonyms allow to use several expressions to designate the same object.

Since the first version of PDM, the vocabulary has been somewhat normalized. Since we are mostly interested in representing knowledge, we modified our own jargon to meet current use. Thus, the following changes have been introduced:

Previous name	**New name**
Entity	Concept
Terminal property	Attribute
Structural property	Relation
Instance	Individual

2 Some History

The PDM model is the result of a bottom up approach in which classes are abstracted from the existing representation of individual objects, and used as a soft specification for such objects. Individual objects do not belong necessarily to a class, and we use a mechanism similar to that proposed by Stein. Many object models have been proposed in the past both in artificial intelligence and in the database area, and more recently for developing ontologies in the context of the semantic web. In artificial intelligence models tend to act as prototypes and repositories for default values. Many models have been used for representing knowledge in languages like KRL [BW77], [GL80], UNITS [FK85], SRL [FWA86], AIRELLE [ANVPR88], SHIRKA [Rec85], CML [Sta86], and many others that have frame editor capacities.Within the database community attempts were undertaken in the eighties to cope with the need to support efficiently deductive databases and applications like CAD. Object models were proposed like [YOMF86], or O2 [LRV87]. In the database perspective object models tend to act as specifications for storing information.

In object-oriented programming languages the object representation is usually rather poor compared with what one can find in artificial intelligence, with however some interesting exceptions like LORE [BCP86] which used a set-theoretic object definition. Many papers, like one by Cointe [Coi87] tried to improve on the available object format.

[1]We encountered the problem in the TerreGov IST project (2004-2007), for which we developed the SOL syntax.

Within our MOSS environment, objects are represented using an extended version of a model called PDM (Property Driven Model) which is a compromise between a frame model and a semantic data model (see Abrial [Abr74]). PDM like RLL is reflective (see Maes [Mae87] for a discussion about reflection). PDM has both specification (definitional) and prototyping capabilities together with some other interesting features described in the following paragraphs.

3 Basic Object Format

3.1 Properties

A PDM object is described by a set of properties, e.g. (see Fig.1)

```
name:    Einstein
first name:  Albert
birth country: Germany
specialty: Relativity
```

The two first properties have values (i.e. literals) directly attached to them and are called **attributes** (`att`), the two last ones are links to other objects (i.e. an object keyword for specialty and an object country for Germany) and hence are called **relations** (`rel`)[2]. The printed values Germany and relativity are actually a summary description of the corresponding objects.

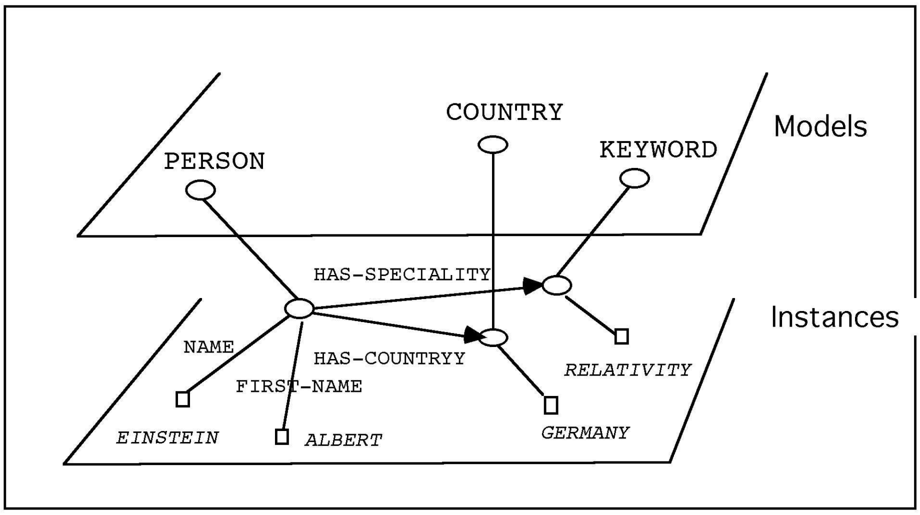

Figure 1: EINSTEIN Example

Objects are instances of classes as shown Fig.1. In addition, but not shown in Fig.2, all structural properties (links between objects) are inverted, so that the object "`germany`" is linked to the object "`einstein`" by the property `is-country-of`.

The reader will have noticed that PDM does not use the classical 3-level frame-slot-facet format found usually in frame languages. Instead properties are themselves declared as objets and described using the same format, e.g.

name	
entity:	Concept
property-name:	NAME
is-attribute-of:	PERSON
method:	=if-added, =print-value, =entry

A relation is described in the same fashion:

[2]In the OWL jargon attributes are called datatype properties and relations are called object properties.

specialty	
property-name:	SPECIALTY
is-relation-of:	Person
successor:	KEYWORD
inverse:	is-specialty-of
method:	=if-added

A long time ago, it was already mentioned by Cointe [Coi87] that such a reification of attributes would be very valuable in SMALLTALK.

3.2 (Soft) Classes

Classes are structural abstractions of instances. The class PERSON is described as follows:

Person	
entity-name:	Person
attribute:	has-name, has-first-name
relation:	has-specialty, has-birth-country
method:	=summary, =print-self,...

On the above examples it can be seen that information usually found in slots is factored in the attribute objects.

The class PERSON is represented Fig.2 in a layer, called models, containing class and property objects. The model layer is a specification layer describing the format (structure) of objects in the instance layer, as shown by the dashed lines on Fig.2.

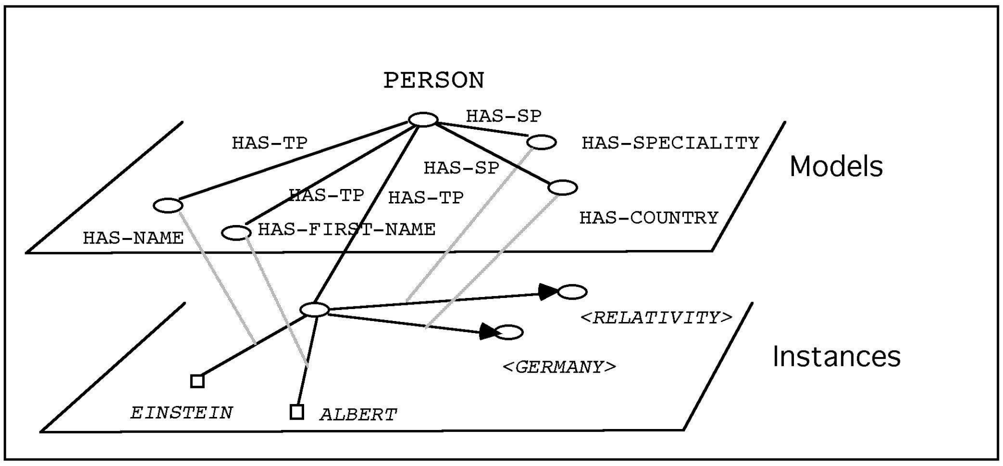

Figure 2: Details of the model specification layer

In the PDM model an individual object (instance of a concept) need not obey the structural definition represented by the concept fully. Hence the term **Soft Class**.

3.3 Orphans

In addition to their properties all the previous objects has a special property called HAS-TYPE to indicate that they can be considered as instances of this class. E.g.

Einstein	
type:	Person
name:	Einstein
first name:	Albert
birth country:	Germany
specialty:	Relativity

Some objects are special and do not belong to any class. They are called **orphans**. This happens for example when a robot encounters an object and does not know what kind of object it is, but acquires progressively more information. The object will remain an orphan until it has enough properties to be classified.

3.4 Defaults and Ideals

Properties (attributes and relations) can be used to specify default values. For example a typical PERSON may be called Smith, be a female and have age 40 and live in New York.

Attribute defaults are specific values (literals) and relation defaults are objects (instances or classes). Defaults can be attached to a special (virtual) instance of a class called an **ideal**. Alternatively, defaults can be attached to attributes directly (useful for orphans), e.g., `HAS-NAME` may record the default ?Smith.? Defaults attached to ideals have priority over defaults attached to properties.

3.5 Object Identifiers

Each object has a unique identifier, based on the radix of the class that defaults to the internal name of the class. E.g., `$E-PERSON.23`, 23 being the value of an instance counter at the time the instance was created. An object thus may be "empty" meaning that no attribute or relationship has any value, but nevertheless the object exists. Classes and properties are objects. The internal identifier for a class by default is built from its name being prefixed by `$E-`. The identifiers for attributes and relationships are built in the same fashion with respective prefixes of `$T-` and `$S-`.

Orphans have identifiers like `$0-12`, whose structure will be explained later.

Ideals have an identifier like an instance but numbered 0, i.e., `$E-PERSON.0`. They may or may not exist for a given class.

3.6 Inverse Relationships

In the PDM format all relationships (structure properties or links) are inverted. Thus, if a person has a brother, the relationship will be inverted to indicate that the other person is the brother of the first one. This allows to traverse the structure in any direction. MOSS maintains all inverse links automatically. Note that an inverse relationship is derived from the direct relationship and has a different semantics (more on this point later).

3.7 Inverse Attributes: Entry Points

Attributes can also be inverted and point onto a special object called an **entry point**, used as an index to locate the object. For example, the name ?Einstein? could be inverted and become an entry point. An entry point is an object whose identifier is the index, e.g. `EINSTEIN` in the previous case. The inverted name property (`IS-NAME-OF` internally `$T-NAME.OF`) points to the identifier of the Einstein object. Note that ?`Einstein`? could be the name of several persons or even the name of a company. Since our properties are multivalued, this is taken into account.

3.8 Demons

Inherited from artificial intelligence is the concept of **demon**. Demons are special functions (or methods) that are triggered automatically on special events. Several demons are very useful: `xi`, `if-added`, `if-removed`. Their functioning will be presented later.

3.9 Models and Meta-Model

Just as objects could be structurally abstracted to give birth to classes, classes and properties, that are objects, can be abstracted in turn as shown Fig.3.

Thus attributes (terminal properties in the figure), relations (structural properties) or concepts (classes in the figure) have models (superclasses), and, in turn those objects can be abstracted into a meta-meta-object that we call **entity**. To the question can the object entity be abstracted in turn,

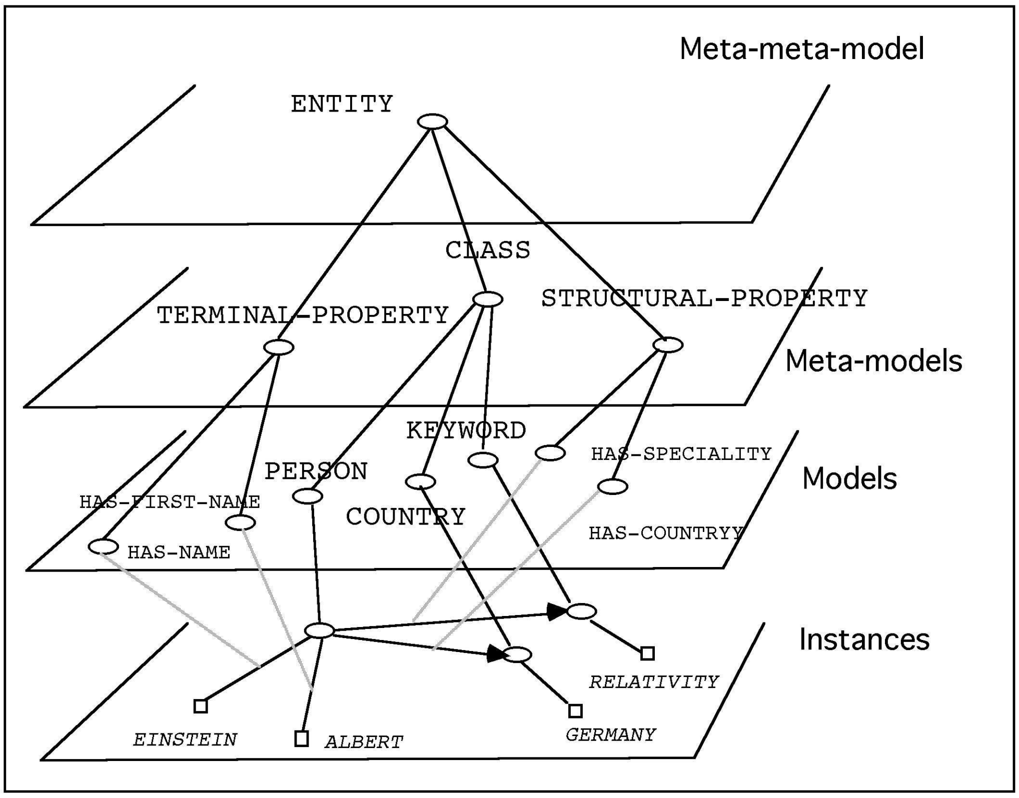

Figure 3: PDM specification hierarchy

the answer is ?yes, of course.? However, entity happens to be its own model. Thus, *PDM is reflective* (it contains its own model) as many frame languages in artificial intelligence.

3.10 Value Inheritance, Prototyping

PDM offers the concept of **prototype**. Object A can be declared to be a prototype for object B. In that case, object A will provide all values that are not specified locally for B. Thus, if A and B are two objects and the shape of B is not known, then B will inherit the shape of A, a value. If B has a value, the value will shadow that of A.

Prototyping is useful when an object has no class but is known to look like or to behave as another object in the system.

3.11 Worlds, Views and Versions

PDM objects can be versioned, i.e., they can exist in different worlds or be looked or from different viewpoints. Versions in PDM are called **contexts** or **versions**. It is implemented as a world mechanism as for example in the former ART environment. It helps to solve problems like "Mary is 24 but Joe believes she is 22." PDM will create two worlds, one for the official numbers, and one for Joe.

All objects without exception can be versioned.

4 Behavior, the MOSS environment

The PDM model has been implemented in an environment called MOSS as an (extended) object-oriented language. Thus, MOSS implements message-passing, has methods and (extended) inheritance. The implementation was done in ANSI Common Lisp.

4.1 Message Passing

Objects can send or receive messages using the send function, e.g.

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

where the variable cb contains an object identifier (e.g. $E-PERSON.4). This yields:

```
----- $E-PERSON.4
 name:  Barthes
 first name:  Camille, Xavier
 sex: m
 sister: Peggy Barthes
-----
:done
```

A broadcast function can be used to send the same message to a number of objects. However, it does not execute in parallel.

4.2 Methods

Like in any OOL we use methods. PDM has three kinds of methods that are all first class objects: instance methods, own methods and universal methods.

4.2.1 Instance Methods

Instance methods are similar to methods in standard OOL, like Java. They are attached to classes and apply to instances of the class.

4.2.2 Own Methods

Own methods are attached directly to an object and can be used to specify the behavior of orphans, to express creation methods (like new) for classes, or to express exceptions.

4.2.3 Universal Methods

Universal methods are default methods that are not attached to any object in particular. They apply to all objects. Methods like `=print-self`, `=get-properties`, methods handling the object structure are universal methods.

4.3 Inheritance

PDM classes may have super-classes. However, PDM handles inheritance somewhat differently from standard OOL, due to the presence of prototypes (e.g., Fig.3) and of the different types of methods. When an object receives a message, it first checks if the corresponding method is an own-method, or if it can be inherited as an own method along an `IS-A` link. If not, then it asks its class for an instance method, either directly attached to its class or inherited via an IS-A link in the class specification layer. If not, then it checks if the method is not a universal method, in which case it is readily executed. If no method is found, then MOSS sends the object a message "`=unknown-method`" (which is a universal method).

Let us stress that a method is an object and as such can be versioned or can have methods.

4.4 MOSS Kernel

A number of classes and of methods have been defined to build the MOSS environment. Classes define the object structure, methods define the API. They are described in a separate document.

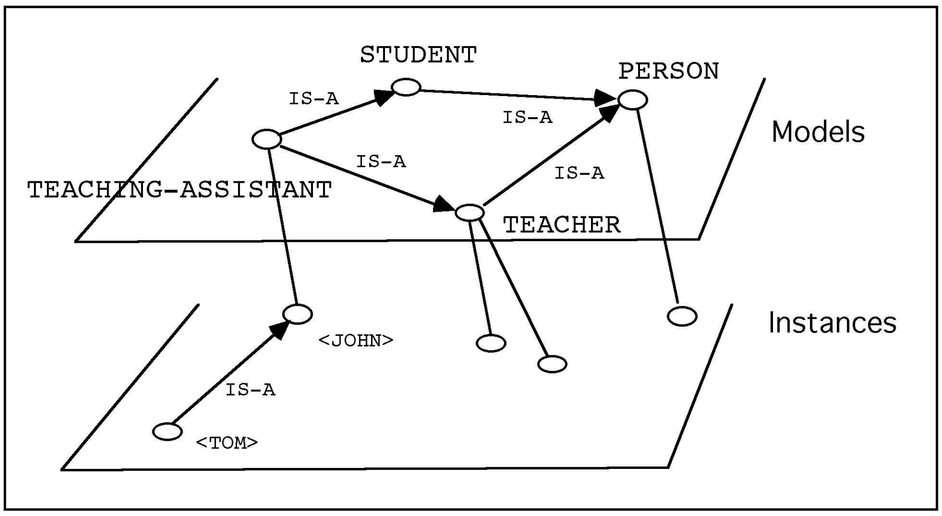

Figure 4: Inheritance hierarchy within a specification layer

5 PDM as a Representation Language

A question that must be discussed is the following: PDM is an object model and MOSS is an Object-Oriented Language. But are the features presented in the previous sections good enough for a representation language, and in particular how is reasoning implemented?

A representation formalism should be discussed from two points of view: its expressiveness and its inference capabilities. We start with the second subject and will address the first one in Section 5.2.

5.1 Inference

PDM allows different types of reasoning. One is linked to the inheritance of values or methods. Another to the possibility of querying the system, a third one to the execution of methods, and a fourth one to the possibility of defining rules and executing them.

5.1.1 Defaults

Defaults are part of the PDM model and of the MOSS system. They are specified using the Ideal mechanism or at the property level. Noting special must be done.

Demons fall in the same category.

5.1.2 Queries

A query language, OQL, has been defined to extract subsets of objects from the system. Its syntax is fairly complex and detailed in a separate document.

5.1.3 Methods

As with any OOL methods can be written without any limit on what could be expressed.

5.1.4 Rules

As with some of the former frame languages (e.g., KEE or KC) one can define rules and executes them on the data, implementing standard expert systems, with various controls. One could also (but this is not implemented) define a method for doing PROLOG style resolution. This is used in some applications mostly for local bookkeeping rather than as an overall reasoning method.

5.2 Expressiveness

MOSS as it is, allows duplicating most of the mechanisms found in the various OOL (with the exception of multi-methods of CLOS), with additional features found previously in frame languages. This however does not confer a high representational power.

5.2.1 Extensional Representation

PDM objects represent actual objects whether concrete or abstract. The PDM representation is thus an extensional representation. A PDM class is not an intentional representation of the concept of PERSON for example, but a specification of the typical structure of instance objects. Thus, as it was defined intentional information must be expressed as methods to be applied (sometimes automatically, e.g. for demons) to the extensional representation of objects. Intentions buried within the code of methods are not available for reasoning.

5.2.2 Intentional Representation

We intend to add intentional capabilities to PDM by defining intentional objects, namely:

- An object representing the set of objects for a given class (this was already present in NETL);
- An object for representing some subset of objects of a class;
- An object representing a single unspecified object of a class.

This approach allows representing predicative formulae easily, but interferes with the previous mechanisms (e.g., the query mechanism), and thus needs more work

6 Database Features

Several extensions were done in the past for saving the objects onto disk and allowing multi-user access (LOB and VORAS environments). However, we currently use the language to model ontologies in multi-agent systems, and those particular aspects not being so crucial have been removed to simplify the MOSS environment.

7 Multilinguism

Developing ontologies across different languages implies to be able to give different names in different languages, for concepts, but also for individuals. However, using the ontology is done within the context of a particular language. Thus, outputs should respect this approach.

7.1 Specifying Multilingual Names

Specifying names in different languages is done by using a new data type: a multilingual name; expressed as a list including word strings with language tags, e.g.

```
(:name :en "person" :fr "personne")
```

where the first tag(:name) is optional.

More formally:

```
<Multilingual-name> ::= ({:name} {<tag> <synonyms<}+)
<tag> ::= :en | :fr | :it | *
<synonyms> ::= "<text> {; <text>}* "
<text> ::= any string not containing " without escape \
```

When entering data, one can use the format with the provision that language tags are legal, i.e. defined at the system level.

7.2 Allowed Languages

Allowed languages are defined at the system level and have a corresponding language tag (e.g. `:EN` for English, `:FR` for French).

7.3 Current Language

When working in a specific environment, a current language must be specified. The current language filters name for display. The value of a multilingual name in a name whose tag it does not contain is "?".

7.4 Default Language

Default language is English. This is the language of the kernel.

7.5 Using Multilingual Names

Multilingual names may be used anywhere in inputs. Mixing languages is allowed.

8 Synonyms

8.1 Multilingual Name

Defining ontologies requires using synonyms. Synonyms are allowed when specifying multilingual names by using semi-columns in name strings, e.g.

```
(:en "person ; guy" :fr "personne ; quidam")
```

The number of synonyms is unlimited and spaces are not required before or after a semi-column.

8.2 Using Synonyms

Synonyms may be used for defining concepts, properties or individuals, e.g.

```
(defindividual "quidam" ("nom" "Albert"))
```

Synonyms can be used anywhere, and languages can be mixed in inputs, e.g.

```
(defindividual "guy" ("nom" "Joe"))
```

Each synonym generates an entry point (index object).

9 Generic Properties

Properties are created as independent objects, called **generic properties**. For example, a property ?name? will be created as a generic property independent from any class reference. If a class refers to that property, then a new property, attached to the specific class will be created, bearing the same name. Thus, a person will have a `"name"` corresponding internally to a `"person-name"`; a bookstore may have a `"name"` corresponding to a `"bookstore-name"` maybe with a different behavior, like different cardinality constraints. MOSS handles the differences automatically.

Generic relations are defined as relations from any object to any object (class ***any***), including classes or individuals.

10 One-Of

The ”one-of” construction is intended for constructing a class from a list of instances, meaning that the class comprises such instances and no others.

This does not fit the semantics of MOSS classes (concepts) since they are only meant to abstract the structure of a set of objects and not give the list of allowed objects, which represent a necessary and sufficient condition.

When applied to a class, the list of instances can be stored at the class level, associated with a `$TYPE.OF` property. The presence of such a property does not block make-instance functions. Indeed, the user in the MOSS philosophy is entitled to create objects that violate constraints, whenever necessary. MOSS issues a warning.

When applied to a property, then we consider two distinct cases: attribute and relation.

In the attribute case, a one-of indicates a series of possible choices for the value associated with the attribute. Thus, it can be associated with the property `$VAL`.

In the relation case it contains a list of possible successors of the specific property, i.e. the class of successors is an anonymous class made of the set of individuals specified by the `:one-of` option. If a `:to` property is also present then the individuals must be of the type specified by the `:to` option (range).

11 Restrictions

Additional restrictions may be imposed locally, i.e. to a property associated with a class. They are specified by the `:exist` and `:forall` options.

The `:exists` option indicates that among the values (for an attribute) or successors for a relation), one at least of the item must be taken from the :one-of list or be of the specified type when the `:one-of` option is not present. There must be at least one value.

The `:forall` option indicates that if there are items associated with a property, all of them must be taken from the one-of list or be of the specific type (when the one-of option is not present). There may be no value.

A special `:not` option indicates that values or objects should not belong to the one-of list.

Other operators are available for attribute values, namely, `:between` and `:outside`.

Cardinalities are treated as restrictions.

12 Multiple Types

An individual can belong to multiple classes, inheriting properties from each class. Like in CLOS, classes are ordered in such a way that properties inherited from classes in front of the list shadow properties inherited from other classes.

The internal implementation for selecting the object id selects the first class as the ”primary index” and the other classes as ”secondary indexes” or ”reference-ids,” introducing a level of indirection for addressing the object. Thus any object may be a reference-id meaning that it must be de-referenced to access its actual value.

References

[Abr74] Abrial. Data semantics. In J.W. Klimbie and K.L. Koffeman, editors, *Data Base Management*. North Holland, 1974.

[ANVPR88] Anne Adam-Nicolle, Bernard Victori, Christine Porquet, and Marinette Revenu. Airelle, rapport de présentation. Technical report, Université de Caen, Fevrier 1988.

[BCP86] Christophe Benoit, Yves Caseau, and Chantal Pherivong. The lore approach to object-oriented programming. Technical Report Memo C29.0, CGE, Laboratoires de Marcoussis, April 1986.

[BW77] Daniel G. Bobrow and Terry Winograd. An overview of krl, a knowledge representation language. *Cognitive Science*, 1(1):264–285, 1977.

[Coi87] Pierre Cointe. Metaclases are first class: the objvlisp model. *Special Issue of SIGPLAN Notices*, 22(12):441–451, 1987.

[FK85] R. Fikes and T. Kehler. The role of frame based representation in reasoning. *Communications of the ACM*, 28(9):904–920, 1985.

[FWA86] Mark S. Fox, Mark Wright, and David Adam. Experiences with srl: An analysis of a frame-based knowledge representation. In L Kerschberg, editor, *Expert database systems*, pages 161–172. The Benjamin/Cummings Publishing Company Inc, 1986.

[GL80] Russel Greiner and Douglas B. Lenat. A representation language language. In *AAAI-80*, pages 165–169, Stanford, 1980. Stanford University.

[LRV87] Christophe Lécluse, Philippe Richard, and Fernando Velez. O2, an object-oriented data model. Technical Report Altaïr, INRIA, 1987.

[Mae87] Patty Maes. Concepts and experiments in computational reflection. *Special Issue of SIGPLAN Notices*, 22(12):147–155, 1987.

[Rec85] François Rechenman. Shirka: Mécanismes d'inférence sur une base de connaissances centrée objets. In *5e Congrès AFCET sur la reconnaissance des formes et l'intelligence artificielle*, Grenoble, France, 1985.

[Sta86] Stanley. *CML: A Knowledge Representation Language with Applications to Requirements Modeling*. PhD thesis, University of Toronto, 1986.

[YOMF86] Xu Yu, Nobuo Ohbo, Takashi Masuda, and Yuzuku Fujiwara. Database support for solid modeling. *The Visual Computer*, 2(6):358–366, 1986.

Index

broadcast function, 12

database, 14

entry point, 10
exist option, 16

forall option, 16

generic properties, 15

ideal, 10
identifiers, 10
inheritance, 12
intentional representation, 14

kernel methods, 12

language tags, 15

multilingual names, 14
multiple concept belonging, 16

orphan, 10

prototype, 11

rules, 13

send function, 12
soft class, 9
synonyms, 15