UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique

MOSS 7 - Query System

Jean-Paul Barthès

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

Email: barthes@utc.fr

N222
July 2008

Warning

This documents describes the MQS of the MOSS 7 query system.

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, query system

Revisions

Version	Date	Author	Remarks
0.1	Jan 95	Barthès	Draft MOSS v3.2
1.0	Aug 04	Barthès	Extension to MOSS v4.2
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

1 Introduction

What is a query system in an object system?

A query system is a piece of software that allows to access objects according to some criteria, e.g. "all persons who have a red car." It is an essential part of any database or knowledge base system.

For some object systems the query system has to face a serious problem. Indeed, objects are commonly spread randomly across the disk area, and for very large object bases one needs to perform many disk reads; however, disk reads are expensive. It is therefore crucial to minimize the number of such disk reads by organizing the sequence of accesses cleverly. This was already the case for the OQA system processing all requests onto the VORAS prototype. The OQA system was improved by Li Yi (Li Yi 89), who introduced the possibility of having OR and NOT operators in a query, and processing the query without having to reduce the query logically to conjunctive normal form, nor to disjunctive normal form. Indeed, while doing so, one tends to increase the number of redundant disk reads.

The current version of the query system was changed in version 3.2 to improve the expressiveness of the allowable requests. The system was first upgraded to MOSS v4.2. This document describes the upgraded version to MOSS v7, which is not very different from the previous one.

Warning In this document we use indifferently the terms class or concepts, and individual or instance.

2 Overview of the MOSS Query System

The next lines describe the overall approach I adopted for MQS (MOSS Query System).

2.1 Input Query Format

Complex queries are defined as a tree in the graph of models defining the object base.

Examples:

```
Persons whose name is Dupond
("person" ("name" :is "Dupond"))

Persons whose name is Dupond, and who have a brother whose age is greater than 25
("person" ("name" :is "Dupond"))
  ("brother" ("person" ("age" > 25))))

Persons whose name is Dupond, and who have a brother whose age is greater than 25,
and who is not employed in a company not named UTC...
("person" ("name" :is "DUPOND"))
  ("brother" ("person" ("age" :between 25 30)))
  ("employer" ("company" ("name" :is-not "UTC")...)
```

Queries can be arbitrarily complex, provided they retain the form of a tree.

2.2 Basic Query Mechanism

The query system takes a formal query as input, focusing on a particular concept. It then computes a set of possible candidates, and tries to validate each candidate in turn by navigating *à la Prolog*.

2.3 Semantics of the Recorded Values

For the time being, MOSS does not record null values. A null value associated with a given property (attribute or relationship) means that MOSS does not know the value associated with the property (most of the time the property is not even recorded in the object). Thus there is no way of specifying currently that a property is known and that there is no value associated with it. For example, some persons from India have a family name but no first name. Thus we should store the attribute "first name" with a null value, meaning that there is no first name. This is currently not the case.

Thus, the answer to a query means that the objects are those known to MOSS to match the query constraints.

2.4 Implementation

The approach is implemented in three steps:

1. the formal query is parsed and replaced with a query using internal symbols, the syntax being checked for correctness;
2. a candidate list is computed using entry points in the query;
3. objects of the candidate list are checked in turn against the full query for validation. The result is a list of object identifiers.

A *query handler* object is defined to implement the query mechanism. Two MOSS methods?associated with the query handler?are available, depending on whether the user wants an interactive control, or a programmed?automatic?execution.

3 MOSS Query Syntax

The general query format is the following:

```
<general-query> ::= <entry-point> | <query>
<query>         ::= (<class> {<clause>}*)
<clause>        ::= (<simple-clause>) | (<or-clause>)
                      | (<or-constrained-sub-query>)
<simple-clause> ::= <tp-clause> | (<sub-query>)
<tp-clause>     ::= (<terminal-property> <tp-operator> <value>)
                      | (<terminal-property> <tp-operator> <variable>)
<or-clause>     ::= (OR <simple-clause>+)
<mixed-clause>  ::= (<sub-query>) | (<simple-clause> . {<mixed-clause>})
                                  | (<sub-query> . <mixed-clause>)
<sub-query>     ::= (<relationship> <cardinality-condition>  <query>)
<relationship>  ::= <structural-property> | <inverse-property>
<or-constrained-sub-query> ::= (OR <cardinality-condition> {<sub-query>}+)
<cardinality-condition> ::= (<numeric-operator> <arg>+)
<numeric operator> ::= < | <= | = | >= | > | <> | BETWEEN | OUTSIDE
<tp-operator>    ::= < | <= | = | >= | > | ? | BETWEEN | OUTSIDE
                         | IS | IS-NOT | IN | ALL-IN
                           | CARD= | CARD< | CARD<= | CARD> | CARD>=
<variable>       ::= symbol starting with ?
<entry-point>    ::= symbol | string

where * means 0 or more, and + one or more
```

3.1 Entry-Point Queries

An entry-point query consists of a symbol or a string. For example:

```
Asking for any object that has an entry point of DUPOND
DUPOND or "Dupond"
```

As an answer to this question, MOSS will simply collect all objects that have DUPOND as an entry-point (obtained from applying the default make-entry-point to the symbol or the string) of any possible attribute with entry-points. It will return a list of such objects.

3.2 Class Queries

This sort of query is used to obtain all elements from a given class. E.g., all persons:

```
(person) or ("person")
```

MOSS will return all elements of the class PERSON (individuals corresponding to the concept PERSON, including all elements of the subclasses of persons, e.g., teachers, students, etc.

One can disable getting information from subclasses by setting a global variable (see advanced features).

3.3 Simple Queries

The following queries are the simplest possible queries, e.g.,

- All males

  ```
  ("person" ("sex" :is "m"))
  ```

- All persons named "Barthès" and known not to be males

  ```
  ("person" ("name" :is "Barthès") ("sex" :is-not "m"))
  ```

  Notice that putting several clauses amounts to having an implicit AND on such clauses.

- All persons with brothers

  ```
  ("person" ("brother" ("person")))
  ```

  For such questions which do not include a cardinality constraint following the relationship, the parser adds a default condition (> 0), so that the query becomes:

  ```
  ("person" ("brother" (> 0) ("person")))
  ```

  When MOSS processes the query, it starts with a set of possible candidates and examines each candidate in turn. For each candidate, it follows the has-brother relationship and is happy as soon as it finds a reference to a brother.

3.4 Queries with a Longer Path

This is the same as above but the query contains a longer path.

- persons who are cousins of a person who has a father with name "Labrousse"

```
(person (is-cousin-of
           (person (has-father
              (person (has-name :is "Labrousse"))))))
```

Note that we used the names of the concept and of the properties directly. When using strings an inverse property is written as "¿cousin". Hence we could write:

```
("person" (">cousin"
           ("person" ("father"
              ("person" ("name" :is "Labrousse"))))))
```

- persons who have a brother who is cousin of a person who has a father with name "Labrousse"

```
(person (has-brother
       (person (is-cousin-of
            (person (has-father
                  (person (has-name :is "Labrousse"))))))))
```

3.5 Queries with Disjunctions

- All persons who have at least a brother, or a sister

```
(person
     (OR  (has-brother (person))
          (has-sister (person))))
```

- All persons who have at least a daughter or a son, and a brother

```
(person
     (OR  (has-son (person))
          (has-daughter (person)))
     (has-brother (person)))
```

Note that all queries are set in conjunctive normal form. I.e., they appear as a conjunction of clauses, each clause being simple or a possible disjunction.

3.6 Queries with Cardinality Constraints (Including Negation)

The first example is a way to express negation:

- All persons with no (gettable or recorded) brothers (close world)

```
(person (has-brother (= 0) (person)))
```

The next examples include some sort of restrictions:

- All persons who have only one brother called "Sebastien"

```
(person (has-brother (= 1)
           (person (has-first-name is "Sebastien"))))
```

- All persons, brother of a person called "Sebastien" (inverse relationship)

```
(person (is-brother-of (= 1)
           (person (has-first-name is "Sebastien"))))
```

- All persons who have a father with more than 2 boys

```
(person (has-father (person (has-son (>= 2) (person)))))
```

4 Query Structure

A query is formulated as a tree, the nodes of which correspond to sets of objects?classes. A query can be represented as a *constrained graph*. Constraints are expressed as filters attached to the nodes. Except for queries that consist of a single entry point, all queries start with the name of a class. For example:

```
(PERSON (HAS-NAME :is "Dupond")
        (HAS-FIRST-NAME :is "Jean")
        (HAS-SISTER (PERSON (HAS-AGE < "20"))
        (HAS-EMPLOYER (COMPANY (HAS-ABBREVIATION :is "ABC"))))
```

The corresponding constrained graph has 3 nodes: the first one is the goal node corresponding to the objects answering the query; the second is a node for the persons that can be sisters of the persons we are looking for; the third one is a node for companies (Fig.1).

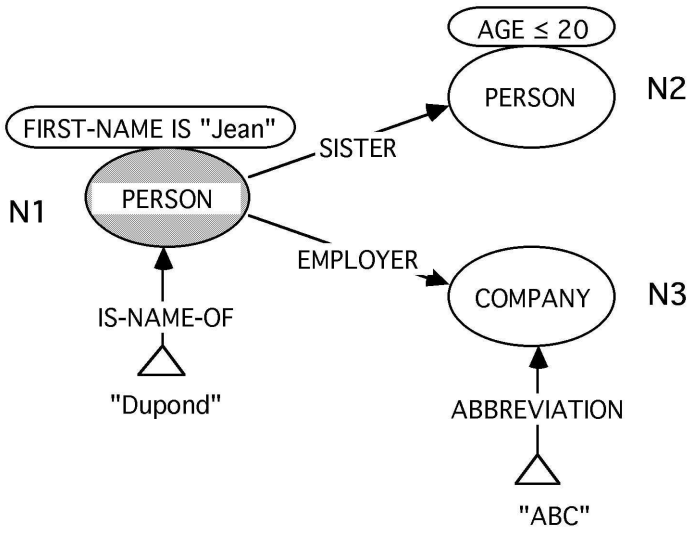

Figure 1: Node structure showing the goal node N1 and the related nodes: N2, N3. Constraints are shown in ovals, entry points are shown using triangles.

The goal is to find all persons named "Jean Dupond" with a sister less than 20, and who work for a company named ABC. To do so, one could select several strategies:

- S1: look at all persons in turn, select those who are named Dupond, and whose first name is Jean, then for all such persons check that they have a sister who is less than 20, then look for those who work in a company named ABC.
- S2: look at all persons who are less than 20, then look at all the persons who have such persons as sisters, then in the last lot check for those who work for ABC.
- S3: look at all employees of ABC, then keep those who have a sister who is less than 20.

Clearly, if our environment contains several thousands of persons, then strategy S1 is not very efficient, and it would be better to use strategy S2 or S3. In fact, here, "Dupond" and "ABC" are entry points and can be used to select only persons whose name is "Dupond" or companies called "ABC".

Access Mode: In practice MOSS uses the entry-points to first select a subset of the class of persons, those that are named "Dupond" and who work for the company "ABC". Then, for each person in this starting subset, the full query is tested, i.e. the first-name must be "Jean", and the person must have a sister with age less than 20. Only objects satisfying all conditions are put into a candidate list.

Each object of the candidate list is checked progressively, using the various links. On this example we would check first that a given person Pi in the candidate set has "Jean" as a first name, then has a sister with age less than or equal to 20 (we stop checking as soon as we have found a sister satisfying the constraint, or if there is no such sister, in which case the query fails for Pi and Pi is discarded), then is working for a company names "ABC".

5 Access Algorithm

The purpose of the algorithm is to minimize the number of accesses to objects that must be visited.

At a given node access to objects can be done locally, either by means of entry-points if there are any, or by enumerating the objects in the class and subclasses (default behavior) represented by the node, or via the neighboring nodes when the query contains subqueries. However, in the latter case, access by means of inverse properties can only be done when the cardinality constraint is positive, i.e. (> 0) or ($>= 1$), meaning that there should be at least one neighbor (default). When the cardinality constraint is different, e.g. ($= 0$) or ($= 2$) or (> 3) or ($BETWEEN25$), then such constraints must be used a posteriori and access must be done in the first node. The case of OR clauses is more complex.

5.1 Algorithm

When the query reduces to a single entry point then the answer is obtained immediately. Otherwise, we use the following algorithm.

Step 1 - Estimate a list of candidates

Use the query entry points in positive branches to compute a subset of the class of the goal node. The result may be *none* meaning no answer, a list of candidates, or nil, meaning that we must do a class access (i.e. load all the objects of the class).

Step 2 - Collect objects

For each object in the candidate list obtained in Step 1, navigate through the constrained graph and apply all filters to determine if the object must be kept in the final solution.

Thus, we start with a list of candidates, and a global test on the subset of objects to be checked. Possible test operators are ALL (for checking all objects), $<, <=, =>, >=, <>$, BETWEEN, OUTSIDE. If the test is verified, then we return a list of the accepted candidates; otherwise we return nil.

For each node of the constrained graph, corresponding to a particular set of objects, we operate as follows:

- First, we split the query into simple clauses?from which we dynamically build and compile a local filter?, and OR-clauses and sub-queries.
- At each cycle, we have a list of candidates waiting to be examined, and a good-list of objects which were candidates and which passed the tests.

  applying the global test to the good-list.

  If we have an immediate success, then we return the set of objects in the good-list.

  If we have a failure, we return nil.

  If we cannot decide, then we examine the current candidate.

  examining the current candidate:

  we first apply the local filter. If the test fails, then we proceed with the next candidate in the candidate list.

  Then, we must check the set of all clauses (conjunction of OR-clauses and sub-queries):

  if there are no more clauses to check, we have a success and we proceed with the next candidate

  if the clause is a sub-query, we proceed recursively.

  if the clause is an OR-clause, we check each sub-clause in turn until we find one that is verified.

6 Constraints on the Relationship Cardinality

Assume now that we want to put some constraint on the cardinality of a relationship, e.g. we want to know the persons named Jean Dupond, who work for ABC, and who have exactly 2 sisters with age less than 20. This can be expressed as:

```
(PERSON (HAS-NAME :is "Dupond")
        (HAS-FIRST-NAME :is"Jean")
        (HAS-SISTER (= 2)(PERSON (HAS-AGE < "20"))
        (HAS-EMPLOYER (COMPANY (HAS-ABBREVIATION :is "ABC"))))
```

The constrained graph in this case is very similar to that of Fig.1 but for the fact that we have now a constraint on the cardinality of the sister relationship.

In this case, we collect sisters as previously and we can distinguish two cases: (i) we find more than 2 sisters with age less than or equal to 20, in which case the corresponding starting object fails to meet the query requirements and is thus removed from the solution list; (ii) in the other case we must wait until we have examined all sisters to check for their number.

7 Queries Containing OR Clauses

OR clauses are defined in a restricted way in MOSS. Indeed, they can involve simple clauses (mix of tp-clause and sub-query) or constrained sub-queries. This section intends to give several examples and relate them to access considerations.

7.1 Simple tp-clauses without entry points

Example

```
(OR (HAS-HAIR-COLOR :IS "Red") (HAS-AGE > 22))
```

It is not possible to use this clause to infer a list of possible objects. The two clauses will be used to filter a list of possible candidates.

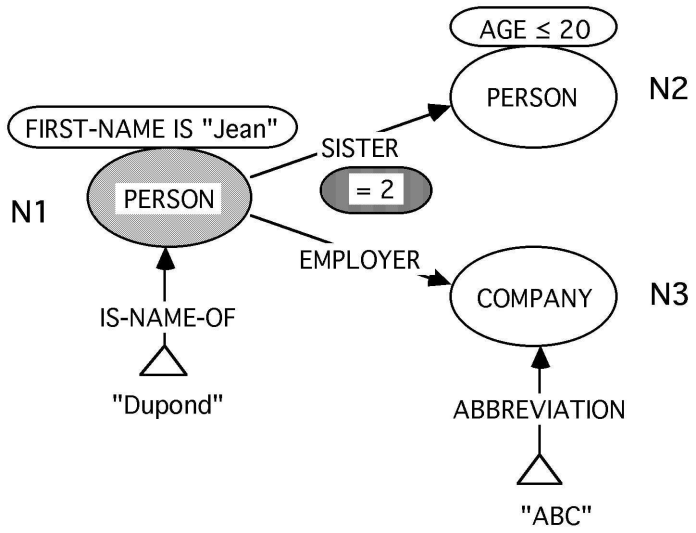

Figure 2: Constrained graph showing an intermediate cardinality filter

7.2 Simple tp-clauses with entry points

Example

```
(OR (HAS-NAME :IS "Dupond") (HAS-COUNTRY :IS "France"))
```

Assuming that NAME and COUNTRY are entry points for a person, means that we can obtain the list of objects whose name is DUPOND and country is FRANCE. Merging the two lists yields a list of possible candidates. Note that we can do this only because each clause has an ?IS? operator.

7.3 Simple tp-clause with and without entry points

Example

```
(OR (HAS-NAME :IS "Dupond") (HAS-HAIR-COLOR :IS "Red"))
```

DUPOND could yield a list of possible candidates. However, HAIR-COLOR brings no specific restriction on the set of possible objects of the class. Thus we cannot retrieve any restricted list of possible candidates from this clause.

7.4 Sub-queries

Example

```
(OR (HAS-BROTHER (PERSON (HAS-NAME :IS "Dupond")))
    (HAS-EMPLOYER (COMPANY (HAS-NAME :IS "ABC")))))
```

In this case, each branch of the OR clause can yield objects. By accessing persons who are brothers of persons named DUPOND and persons who work for the ABC company, we can fuse those two lists that will provide a final list of possible candidates.

7.5 Constrained sub-queries

Example

```
(OR (> 2) (HAS-BROTHER (PERSON))
          (HAS-SISTER (PERSON)))
```

meaning than we can accept any combination of brothers and sisters provided there are more than 3.

Example

```
(OR (> 2) (HAS-BROTHER (= 2) (PERSON))
          (HAS-SISTER (PERSON)))
```

in that case the number of brothers (if any) must be exactly 2.

Only sub-queries can be constrained this way. If we mix sub-queries and tp-clauses in an or-constrained clause the semantics becomes tricky. This MOSS does not allow such mixed constrained OR-clauses.

If the cardinality constraint is positive, i.e. requires at least one value, then we can draw the same conclusion as in 7.4. Indeed, if the clause must be validated, it will necessarily involve some of the objects found in the branches of the sub-queries. Hence, such objects can be used to recover a list of possible candidates by following the inverse-property link.

7.6 Default sub-query constraint

We have shown sub-queries without cardinality conditions, e.g.

```
(OR (HAS-BROTHER (PERSON (HAS-NAME :IS "Dupond")))
    (HAS-EMPLOYER (COMPANY (HAS-NAME :IS "ABC"))))
```

In practice the system adds automatically a default constraint (¿ 0) when there are none, which yields:

```
(OR (HAS-BROTHER (> 0) (PERSON (HAS-NAME :IS "Dupond")))
    (HAS-EMPLOYER (> 0) (COMPANY (HAS-NAME :IS "ABC"))))
```

7.7 Negative OR clauses

Negation is expressed by the (= 0) cardinality constraint.

Example

```
(OR (= 0) (HAS-BROTHER (= 2) (PERSON))
          (HAS-SISTER (PERSON)))
```

meaning that we do not want people who have 2 brothers or sisters. Now if we look at the query at a higher level we might have something like:

```
(PERSON (HAS-NAME :IS "Labrousse")
              (OR (= 0) (HAS-BROTHER (= 2) (PERSON))
                                (HAS-SISTER (> 0) (PERSON))))
```

which is actually equivalent to

```
(PERSON (HAS-NAME :IS "Labrousse")
                  (HAS-BROTHER (<> 2) (PERSON))
                  (HAS-SISTER (= 0) (PERSON))))
```

Therefore, in this case the OR clause can be simplified provided we switch the attribute operators to their complement. This operation is done as a preprocessing step of the formal query.

7.8 Conclusion

When dealing with OR clauses, some conclusions regarding the possible object candidates can be drawn in some specific cases as explained in the previous paragraphs. This is taken into account by the find-subset function. However, most of the time, the clauses and sub-queries will be used a posteriori on a list of candidates generated differently.

8 Multiple Values

Having multiple values in the attributes and relationships introduces some difficulty.

Semantics of Multiple Values

The semantics of a multiple value was not made explicit. One must be careful. A multiple value associated with a structural property or relationship means that the link exists between all objects involved in the relationship; e.g.

```
(...(HAS-COUSIN <person-32> <person-48>)...)
```

means that the person has 2 cousins.

For an attribute, this might be different when one is not careful, e.g.

```
(...(HAS-FIRST-NAME "John" "Julius")
```

has the same meaning as above, but a polyline with 3 vertices expressed as:

```
(...(HAS-VERTEX (0 0) (25 100)(0 20))...)
```

may have a different semantics if it is assumed that the middle vertex is an intermediate point in the polyline, and the other ones are end-points. Here, we no longer have a set, but a structured value. Hence, it should be recorded as a single value being a list of ver-tices and not as a multiple value.

```
(...(HAS-VERTEX ((0 0) (25 100)(0 20)))...)
```

Using a set semantics, we still have problems with negative operators on multiple values e.g.

```
(HAS-NAME IS-NOT "Albert")
```

on a value like

```
("Albert" "Joseph")
```

since the generic operator that builds filters is designed for positive operators, thus makes an implicit OR, while with an operator like not-equal we mean an implicit AND. The situation is even more confusing with range operators like $3 < xx < 45$.

To solve the problem we must specify the semantics of multiple values (which depends on the nature of the associated property).

Multiple values mean alternative values (equivalent values).

Hence when we have a positive operator like :is, it means equal to one of any of the values, when it is :is-not (not-equal), it means equal to none of the values. In case of multiple values we could have more specific operators like NOT-ONE, NOT-ANY, <ALL, <ONE, etc.

The idea is to specify the range of operators to build the overall filtering function

9 Querying the Database Model

Querying the database model is not different from querying the data. E.g., if we want to obtain all classes which have a =summary method, we can use the following query:

```
(entity (has-method
           (method (has-method-name is "=summary"))))
```

10 Programmer's Interface

The programmer's interface uses several classes and methods described in this section. In practice however, one uses the access function.

10.1 Classes

QUERY-HANDLER

The QUERY-HANDLER class was designed to provide the user with a few methods for querying the object base. It has no properties. MOSS uses an instance of query-handler, bound to the global variable `*query-handler*` or `*qh*`.

QUERY

The QUERY class is intended to save queries. Hence queries are reified, and can be referenced at a later stage.

A query has the following attributes:

QUERY-NAME: single value that will be used as an entry point

USER-EXPRESSION: the raw query as given by the user

EXPRESSION: the parsed query, containing internal references.

ANSWER: the answer to the query if the user wants to save it.

10.2 Methods

11 Advanced features

11.1 Disabling Access to Sub-Classes

It is possible to disable access to subclasses, by setting the global variable `*query-allow-sub-classes*` to nil.

11.2 Query with Co-Reference Variables

Such queries have Prolog-like variables:

- All persons who have a cousin of the same sex

```
(person (has-sex :is ?x)
        (has-cousin (person (has-sex :is ?x))))
```

When encountered for the first time, the variable must be on the right side of an equality operator. Other references are unrestricted, e.g.

```
(person (has-sex :is ?x)
        (has-cousin (person (has-sex :is-not ?x))))
```

- All persons who have a son with the same name as the son of his brother

```
(person (has-son  (person (has-first-name is ?X)))
        (has-brother
          (person (has-son
                    (person (has-first-name is ?X))))))
```

- All persons with age greater than 22, having a younger sister

```
(person (has-age > 22)
        (has-age is ?x)
        (has-sister (person (has-age <= ?x))))
```

Danger

The meaning of such queries appear to be intuitively simple to grasp. However, this is not the case as demonstrated by the following example:

```
(person (has-son (person (has-age is ?x))
         (has-daughter (person (has-age <= ?x)))
```

Does this query means in the case when a person has several sons and one daughter that the daughter must be younger than one of the sons, or younger than each son? What about the case when we have several sons and several daughter?

For the time being, we decide, that in a Prolog style, the query will succeed when a person has at least one daughter younger than one of his/her sons

11.3 Class Variables

Class variables are not yet implemented.

A Brief Comparison with SQL

This paragraph is not intended to give a detailed comparison between MQS and SQL used for the relational databases, but to illustrate a few points.

A.1 Major Difference between MQS and SQL

A.1.1 Objects vs. Relations

The major difference is that MQS applies to objects and SQL applies to relational tables (i.e., values of attributes).

MQS selects objects in a database and always returns a set of objects, SQL constructs a new relational table by combining pieces of other tables, according to the relational algebra.

A.1.2 Access vs. access and updates

MQS is used to access objects, SQL can be used to create, to access, or to modify objects.

Thus, in order to be able to compare MQS with SQL, we only take from SQL the accessing part of the language, stripping the modifying and the formatting possibilities. This, in effect reduces the language to a few basic constructs, like:

```
SELECT FROM from-item-list
        WHERE predicate
```

A.2 Example

Consider the following relational tables taken from (Date 87) [1].

Suppliers: s

sno	sname	status	city
s1	Smith	20	London
s2	Jones	10	Paris
s3	Blake	30	Paris
s4	Clark	20	London

Part number: p

pno	pname	color	weight	city
p1	Nut	Red	12	London
p2	Bolt	Green	17	Paris
p3	Bolt	Blue	17	Rome
p4	Scew	Red	14	London

Project number: j

jno	jname	city
j1	Sorter	Paris
j2	Punch	Rome
j3	Reader	Athens
j4	Console	Athens
j5	Collator	London

Shipman: spl

[1] C.J. Date, A Guide to INGRES, Addison Wesley, 1987

sno	pno	jno	qty
S1	P1	J1	200
S2	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S3	P3	J1	300
S3	P4	J2	500
S4	P6	J3	300

The relational tables in fact could represent MOSS objects, where the numbers could be considered to correspond to the MOSS internal identifiers of the objects.

Get supplier names from suppliers who supply at least one red part

SQL

```
SELECT DISTINCT S.NAME
FROM S, SP, P
WHERE S.SNO = SP.SNO
AND SP.PNO = P.PNO
AND P.COLOR = 'RED'

SELECT S.NAME
FROM S
WHERE S.SNO IN
( SELECT SP.NO
  FROM SP
  WHERE SP.NO IN
( SELECT P.PNO
         FROM P
         WHERE P.COLOR = 'RED' )
```

MQS

```
(supplier (has-part (part (has-color is "red"))))
```

In the case of MQS, the names must be further retrieved from the resulting list of suppliers.

Get supplier names for suppliers who supply part P2

SQL

```
SELECT S.NAME
FROM S
WHERE S.SNO IN
( SELECT SP.NO
  FROM SP
  WHERE SP.NO = 'P2' )
```

MQS

```
(supplier (has-part (part (has-number is "P2"))))
```

Suppliers who supply at least 200 parts

SQL

```
SELECT S.NAME
FROM S, SP
WHERE S.SNO = SP.SNO
AND SP.QTY >= 200
```

MQS

```
(supplier (has-shipment (shipment (has-quantity >= 200))))
```

Suppliers who supply at least 200 red parts

SQL

```
SELECT S.NAME
FROM S, SP, P
WHERE S.SNO = SP.SNO
AND S.SNO = P.PNO
AND SP.QTY >= 200
AND P.COLOR = 'RED'
```

MQS

```
(supplier
    (has-shipment
        (shipment (has-quantity >= 200)
                  (has-part (part (has-color is "red"))))))
```

B Implementation

This section gives information about how the query mechanism is implemented.

B.1 Overall View of the Query Process

The query is given to the access function that implements the following algorithm:

1. If the query is a symbol or a string, then it is an entry point and it is processed directly by the access-from-entry-point function.
2. Then, if not already parsed, the query is parsed and transformed into an internal format. If errors are found the query returns nil.
3. A set of candidates is computed from the parsed query by calling the find-subset function. The function uses entry points to collect a set of possible candidates, or if there are no useful entry points in any branch of the query synthesizes the list of instances of the target class.
4. If there are no candidates, return nil.
5. Otherwise, call the function filter-candidates that determines whether a list of candidates (instance of the target class) satisfies the constraints imposed by the query.
6. Remove the dead objects
7. Return the list of remaining objects.

The following sections detail the various steps.

B.2 Entry-Point Queries

Entry points are computed using the make-all-possible-entry-points function that collects all =make-entry methods and apply it to the entry point.

```
? (moss::make-all-possible-entry-points "barthes")
(HAS-BARTHES BARTHES)
```

Here the string "barthes" is interpreted as the possible name of a property of the name of a person. Of course "barthes" cannot be the name of a property. However, the situation is different in the following case:

```
? (moss::make-all-possible-entry-points " person name")
(HAS-PERSON-NAME PERSON-NAME)
```

Then all objects corresponding to the computed entry points are collected by the %%get-objects-from-entry-point function and the list of internal identifiers of the objects is returned.

Examples

```
? (MOSS::ACCESS-FROM-ENTRY-POINT "barthès")
($E-PERSON.1 $E-PERSON.2 $E-PERSON.3 $E-STUDENT.1 $E-PERSON.4 $E-PERSON.5
             $E-PERSON.6 $E-PERSON.7 $E-PERSON.8 $E-PERSON.9 $E-PERSON.10 $E-PERSON.11)

? (MOSS::ACCESS-FROM-ENTRY-POINT "name")
($T-NAME $T-PERSON-NAME $T-ORGANISM-NAME)
```

B.3 Parsing a Query

Parsing a query consists in checking the syntax of the query and transforming it into a query with internal format. This is done by the parse-user-query function.

Examples

```
? (moss::parse-user-query
   '(person (has-name is "barthes")
     (has-brother (person (has-first-name is "antoine")))))

($E-PERSON ($T-PERSON-NAME :IS "barthes")
 ($S-PERSON-BROTHER (> 0) ($E-PERSON ($T-PERSON-FIRST-NAME :IS "antoine"))))

? (moss::parse-user-query
   '(person (">cousin" (person (has-first-name is "antoine")))))

($E-PERSON ($S-PERSON-COUSIN.OF (> 0)
               ($E-PERSON ($T-PERSON-FIRST-NAME :IS "antoine"))))
```

Queries including negations are transformed into queries containing negative operators. In addition, parsing the query includes expanding the virtual classes references.

B.4 Collecting a First Subset of Candidates

This step is performed by the find-subset recursive function. The function returns a list of objects computed from the entry-points eventually found in the sub-query corresponding to its argument. When the function returns nil, then this means that there is no limit coming from the current branch (the function argument), when the function returns the symbol `*none*`, then this means that the current branch has no possible solution.

The function takes into account negative conditions as well as disjunctions.

Example

```
(moss::find-subset '($E-person
                     ($T-PERSON-NAME :is "barthes")
                     (or ($S-PERSON-BROTHER
                          (> 0) ($E-person ($T-person-name :is "barthes")))
                         ($S-PERSON-SISTER
                          (> 0) ($E-person ($T-person-name :is "labrousse"))))))
; returns a list of 7 objects
```

The role of this step is to find a minimal set of candidates to be checked in the next step by using the entry points (index objects) found in the various branches of the query if any.

B.5 Filtering Candidates

This is the crucial test implemented by the filter-candidates function.

The function takes a list of objects, a cardinality test to be applied to the objects once they are filtered and a path from the node representing the class of the objects, and bindings of coreferenced variables. The role of the function is to filter the objects, one at a time, checking that they obey the constraints represented by the path: e.g.

```
($E-PERSON ($S-BROTHER (> 1) ($E-PERSON)))
```

which means that we discard objects that have not at least 2 brothers.

As objects are checked they are added to a good-list. Each time we add an object to the good-list we can check the cardinality conditions for failure (e.g. we want less than 2 cousind and have already found a second one, in which case it is not necessary to access more cousins. However, sometimes we want to access all objects that conform to the path. In that case we simply disallow the early success test and keep filtering all the objects.

Now the question is: what do we return?
Sometimes we want the list of objects (e.g. at toplevel), at other times we simply want to know if the candidates pass the cardinality test, in which case a simple T/nil answer is acceptable. Note that we cannot use the list of objects as an answer to this purpose, since the list may be empty, but the test (= 0) could be satisfied. The idea is to return 2 values: a T/NIL value and the list of objects.

Bindings When we call the function, we may have already a set of instantiated coreference variables. They will be used to check the values. E.g.

```
($E-PERSON ($S-COUSIN (> 1) ($E-PERSON ($T-HAS-NAME :is ?x))))
```

with bindings

```
((?x "Labrousse")(?y "Barthès"))
```

If the path contains simple clauses with variables, then the value of bindings will be changed temporarily for checking the rest of the path. However, on return, the value of bindings is not modified and thus need not be returned. Indeed, we'll check the next object with the same initial bindings as as the previous one.

Node Filters

At each node of the query tree, we must collect objects and apply filters corresponding to the clauses of the query. The function split-query-and-compute-filters prepares the stage. The function splits the query into different parts, and compute a filter from attribute clauses and simple OR attribute clauses to apply to future candidates. If the compiler is available, then compiles the filter.

When a cardinality test is provided, builds another filter to test that will apply to the number of retrieved objects. Those operations should be done only once and not everytime a new candidate is examined, which is the justification for this function.

The arguments to the function are:

- query: sub-query, e.g.

```
($E-PERSON ($T-PERSON-NAME :IS \"Barthes\")
        ($S-PERSON-COUSIN (> 0)
          ($E-PERSON ($T-PERSON-FIRST-NAME :IS \"antoine\"))))
```

- cardinality-filter: something like (> 2) or (:BETWEEN 4 5) or nil

The function returns 6 values:

- name of local filter
- name of cardinality filter (can be nil)
- clauses-with-variables
- or-clauses
- subqueries
- class variable

Examples

```
? (moss::split-query-and-compute-filters
   '($E-PERSON ($S-PERSON-COUSIN (> 0)
                                 ($E-PERSON ($T-PERSON-NAME :IS "Labrousse"))))
   )
(#:LOCAL-FILTER-14023
 #:TEST-14024
 NIL
 (($S-PERSON-COUSIN (> 0) ($E-PERSON ($T-PERSON-NAME :IS "Labrousse"))))
 NIL)

  ;;; when compiler is not available:
? (moss::split-query-and-compute-filters
   '($E-PERSON ($S-PERSON-COUSIN (> 0)
                                 ($E-PERSON ($T-PERSON-NAME :IS "Labrousse")))))
((LAMBDA (MOSS::XX)
   (AND (INTERSECTION '($E-PERSON $E-STUDENT) (MOSS::%TYPE-OF MOSS::XX)) MOSS::XX))
 (LAMBDA (MOSS::XX) (IF (> (LENGTH MOSS::XX) 0) MOSS::XX))
 NIL
 (($S-PERSON-COUSIN (> 0) ($E-PERSON ($T-PERSON-NAME :IS "Labrousse"))))
 NIL)
```

Then filters are applied to the candidates at each node recursively.
We have different types of filters.

B.5.1 Clause Filters

Clause filters apply to an attribute clause, e.g.

```
? (make-clause-filter '$T-YEAR '<= 2002)
((LAMBDA (ZZ)
   (AND ZZ
        (EVERY #'(LAMBDA (YY)
                   (<= (convert-data-to-number YY) '2002))
               ZZ)))
 (=> XX '=GET-ID '$T-YEAR))

? (make-clause-filter '$T-YEAR '= 2002 'some)
((LAMBDA (ZZ)
   (AND ZZ
        (SOME #'(LAMBDA (YY)
                  (= (convert-data-to-number YY) '2002))
              ZZ)))
 (=> XX '=GET-ID '$T-YEAR))
```

B.5.2 Filters for Multiple Values

When dealing with multiple values we must produce the ad hoc filters, e.g.

Examples

```
? (moss::make-local-filter-for-multiple-values
   '("barthès" "biesel")
   '$T-PERSON-NAME
   :is)
((LAMBDA (ZZ)
   (AND ZZ
        (INTERSECTION '("BARTHES" "BIESEL")
                      (MAPCAR #'NORMALIZE-VALUE ZZ)
                      :TEST
                      #'EQUAL)))
 (=> XX '=GET-ID '$T-PERSON-NAME))

? (moss::make-local-filter-for-multiple-values
   '("barthès" "biesel")
   '$T-PERSON-NAME
   :is-not)
((LAMBDA (ZZ)
   (AND ZZ
        (NOT (INTERSECTION '("BARTHES" "BIESEL")
                           (MAPCAR #'NORMALIZE-VALUE ZZ)
                           :TEST
                           #'EQUAL))))
 (=> XX '=GET-ID '$T-PERSON-NAME))

? (moss::make-local-filter-for-multiple-values
   '(22)
   '$T-PERSON-AGE
   :equal)
((LAMBDA (ZZ)
   (AND ZZ (SOME #'(LAMBDA (YY) (= (convert-data-to-number YY) '22)) ZZ)))
 (=> XX '=GET-ID '$T-PERSON-AGE))

? (moss::make-local-filter-for-multiple-values
   '(("barthès" "biesel"))
   '$T-PERSON-NAME
   :all-in)
((LAMBDA (ZZ)
   (AND ZZ
        (EVERY #'(LAMBDA (YY) (MEMBER YY ZZ :TEST #'EQUAL)) '("BARTHES" "BIESEL"))))
 (NORMALIZE-VALUE (=> XX '=GET-ID '$T-PERSON-NAME)))
```

B.5.3 Multiple Filters

Multiple filters act on conjunctions or disjunctions of properties, e.g.

Examples

```
? (moss::make-multiple-filter
     '(($T-PERSON-NAME :is-not "Labrousse")
       ($T-AGE <= 23)))
(((LAMBDA (MOSS::ZZ)
    (AND MOSS::ZZ
```

```
         (NOT (INTERSECTION '("LABROUSSE")
                            (MAPCAR #'MOSS::NORMALIZE-VALUE MOSS::ZZ)
                            :TEST
                            #'EQUAL))))
  (MOSS::=> MOSS::XX '=GET-ID '$T-PERSON-NAME))
 ((LAMBDA (MOSS::ZZ)
    (AND MOSS::ZZ
         (NOT (INTERSECTION '(23)
                            (MAPCAR #'MOSS::NORMALIZE-VALUE MOSS::ZZ)
                            :TEST
                            #'<))))
  (MOSS::=> MOSS::XX '=GET-ID '$T-AGE)))

? (moss::make-multiple-filter
     '((OR ($T-PERSON-NAME :is-not "Labrousse")
       ($T-AGE <= 23))))
((OR ((LAMBDA (MOSS::ZZ)
        (AND MOSS::ZZ
             (NOT (INTERSECTION '("LABROUSSE")
                                (MAPCAR #'MOSS::NORMALIZE-VALUE MOSS::ZZ)
                                :TEST
                                #'EQUAL))))
      (MOSS::=> MOSS::XX '=GET-ID '$T-PERSON-NAME))
     ((LAMBDA (MOSS::ZZ)
        (AND MOSS::ZZ
             (NOT (INTERSECTION '(23)
                                (MAPCAR #'MOSS::NORMALIZE-VALUE MOSS::ZZ)
                                :TEST
                                #'<))))
      (MOSS::=> MOSS::XX '=GET-ID '$T-AGE))))
```

Then, the filters are applied to the candidate, and if the candidate passes it is inserted into a good-list.

B.6 Examples of Queries

This section gives some examples of queries that one can test against the family file.

```
;;; test on family data
;;; macro to print the results
(defMacro pp ()
  `(progn (mapcar #'(lambda (xx) (print (send xx '=summary))) *)
          nil))

(moss::access '(person (has-brother (= 0) (person))))
;;;   access all males
(moss::access '(person (has-sex is "m")))
;;; ===== For the next ones use MOSS window...
;;;   access all person named "Barthès" and not known to be males
(moss::access '(person (has-name is "Barthès")(has-sex is-not "m")))
;;;   access all persons with brothers
(moss::access '(person (has-brother (person))))
```

```
;;;   access all persons with no (gettable or recorded) brothers (close world)
(moss::access '(person (has-brother (= 0) (person))))
;;;   access all Barthes's who have only one brother called "Sebastien"
(moss::access
 '(person (has-name is "barthes")
   (has-brother (= 1) (person (has-first-name is "Sebastien")))))
;;;   access all persons brother of a person called sebastien
(moss::access
 '(person (is-brother-of (= 1) (person (has-first-name is "Sebastien")))))
;;;   access all persons with at least 2 children of any sex
(moss::access
 '(person (or (>= 2) (has-son (>= 0) (person))
              (has-daughter (>= 0) (person)))))
;;; answer should be: jpb dbb pxb chb pb mb mbl ml
;;; the following question removes jpb and dbb from the list. Indeed, one must
;;; have at least 2 sons if any, or 2 girls.
(moss::access
 '(person (or (>= 2) (has-son (>= 2) (person))
              (has-daughter (>= 0) (person)))))
;;; answer should be: pxb, chb, pb, mb, mbl, ml
;;; The following question removes mlb and ml, since we want at most 2 girls.
(moss::access
 '(person (or (>= 2) (has-son (>= 2) (person))
              (has-daughter (<= 2) (person)))))
;;; answer should be: pxb, chb, pb, mb
;;; next query has a bad syntax (cannot include TP-clause in a constrained sub-query)
(moss::access
 '(person (or (>= 2) (has-son (>= 2) (person))
              (has-sex = "f")
              (has-daughter (<= 2) (person)))))
;;; correct syntax should be. However it is not accepted (recursive OR)
(moss::access
 '(person (or (has-sex is "f")
              (or (>= 2) (has-son (>= 2) (person))
                  (has-daughter (<= 2) (person))))))
;;; access all persons whith at least 2 children and most 2 girls and a boy
;;; answer is jpb, dbb
(moss::access
 '(person (or (>= 2) (has-son (<= 1) (person))
              (has-daughter (<= 2) (person)))))
;;;   access all persons with at least 3 children one of them being a son
;;; note that we cannot do that with a single OR clause.
;;; answer is pxb, chb, pb, mb
(moss::access
 '(person (or (>= 3) (has-son (>= 0) (person))
              (has-daughter (>= 0) (person)))
   (has-son (>= 1) (person))))
;;;   persons who are cousins of a person who has a father with name "Labrousse"
;;; answer is: psb cxb ab sb eb
(moss::access
 '(person (is-cousin-of
```

```
          (person (has-father
                   (person (has-name is "Labrousse")))))))
;;;   persons who have a brother who is cousin of a person who has a father...
;;; answer is: psb sb eb ab
(moss::access
 '(person (has-brother
          (person (is-cousin-of
                   (person (has-father
                            (person (has-name is "Labrousse"))))))))))
;;;   persons who have a father with more than 1 boy
;;; answer is: jpb pxb ab sb eb mglb
(moss::access
 '(person (has-father (person (has-son (>= 2) (person))))))
;;;   persons who have  a father with  at least 3 children with at least one girl
;;;           and whose father has a brother
;;; answer: ab, sb, eb
(moss::access
 '(person (has-father
          (person   ; condition on the cardinality of the subset!
           (or (>= 3) (has-son (> -1) (person))
               (has-daughter (> -1) (person)))
           (has-daughter (>= 1) (person))
           (has-brother (person))))))
;;; persons who have 2 names: dbb, mgl
(moss::access
 '(person (has-name card= 2)))
(moss::access
 '(person (has-name is "labrousse")(has-name card= 2)))
;;; persons with more than 1 first name: jpb, cxb, chb (!), psb
(moss::access
 '(person (has-first-name card>= 2)))

;;; ------------------- Queries with coreference varaibles
;;;   person who has a cousin of the same sex
;;; answer should be: cxb ab sb sl cl al
(moss::access
 '(person (has-sex is ?x) (has-cousin (person (has-sex is ?x)))))
;;;   person who has a son with the same name as the son of his brother
;;; illegal query since one cannot set coreference variables in a wub-query
(moss::access
 '(person (has-son  (person (has-first-name is ?X)))
   (has-brother
    (person (has-son  (person (has-first-name is-not ?X)))))))
```

C The Family Knowledge Base

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

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

```
;;;============================================================================
;;;06/08/33
;;; F A M I L Y (File FAMILY-Mac.LSP)
;;;
;;; Copyright Jean-Paul BarthËs @ UTC 1994
;;;
;;;============================================================================

;;; This file creates a family for checking the LOB window system, and the query
;;; system. Thus it requires the corresponding modules.
;;;1995
;;;  2/13 Adding relationships in the family to be able to use the file for
;;;       testing the query system
;;;       Adding other persons not actual members of the family
;;;  2/18 Adding a mechanism for saturating the family links -> v 1.1.1
;;;  2/19 Adding the age property to all persons and birth date (year)
;;;       Changing the version name to comply with the curreunt MOSS version 3.2
;;;       -> v 3.2.2
;;;  2/25 Adding orphans -> v 3.2.3
;;;1997
;;;  2/18 Adding in-package for compatibility with Allegro CL on SUNs
#|
2003
 0517 used to test changes to the new MOSS v4.2a system
2004
 0628 removing ids from the class definitions
 0715 adding :common-graphics-user-package for ACL environments
 0927 changinging the è and è characters to comply with ACL coding...
2005
 0306 defining a special file for mac because of the problem of coding the French
      letters
 1217 Changing the file to adapt to Version 6.
2006
 0822 adding count to the inference loop
|#

#+MCL
(in-package "COMMON-LISP-USER")
#+(AND MICROSOFT-32 IDE)
(in-package :common-graphics-user)
#-IDE
(in-package :common-lisp-user)
```

```
#+(or ALLEGRO MICROSOFT-32)
(eval-when (:load-toplevel :compile-toplevel)
  (use-package :moss))

(moss::mformat "~%;*** loading Family 3.4 Tests ***")

(eval-when (load compile eval)
  (use-package :moss))

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

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

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

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

(defconcept (:en "STUDENT" :fr "ETUDIANT")
  (:is-a "PERSON")
  (:rel (:en "COURSES" :fr "COURS") "COURSE")
  )
```

```
(defconcept (:en "ORGANISM" :fr "ORGANISME")
  (:tp (:en "NAME" :fr "NOM"))
  (:tp (:en "ABBREVIATION" :fr "SIGLE") (:entry))
  (:rel (:en "EMPLOYEE" :fr "EMPLOYE") "PERSON")
  (:rel (:en "STUDENT" :fr "ETUDIANT") "STUDENT")
  )

;;;(defownmethod =make-entry (HAS-NAME HAS-PROPERTY-NAME TERMINAL-PROPERTY
;;;                                    :class-ref PERSON)
;;;  (data)
;;;  "Builds an entry point for person using standard make-entry primitive"
;;;  (moss::%make-entry-symbols data))

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

(definstmethod =summary PERSON ()
  "Extract first names and names from a person object"
  ;(append (HAS-FIRST-NAME) (HAS-NAME))
  (let ((result (format nil "~{~A~^ ~} ~{~A~^-~}" (Has-first-name) (has-name))))
    (if (string-equal result " ")
      (setq result "<no data available>"))
    (list result)))

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

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

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

;;; Instances
```

```
(mp ("BarthËs") ("Jean-Paul" "A") "m" _jpb ("BIRTH YEAR" 1945))
(mp ("BarthËs" "Biesel") ("Dominique") "f" _dbb ("HUSBAND" _jpb)("BIRTH YEAR" 1946))
(mp ("BarthËs") ("Camille" "Xavier") "m" _cxb ("FATHER" _jpb) ("MOTHER" _dbb)
    ("BIRTH YEAR" 1981))
(ms ("BarthËs") ("Peggy" "Sophie") "f" _psb ("FATHER" _jpb) ("MOTHER" _dbb)
    ("BROTHER" _cxb)("BIRTH YEAR" 1973))
(mp ("BarthËs") ("Pierre-Xavier") "m" _pxb ("BROTHER" _jpb)("AGE" 54))
(mp ("BarthËs") ("Claude" "Houzard") "f" _chb ("HUSBAND" _pxb)("AGE" 52))
(mp ("BarthËs") ("Antoine") "m" _ab ("FATHER" _pxb)("MOTHER" _chb)
    ("COUSIN" _cxb _psb)("AGE" 27))
(mp ("BarthËs") ("SÈbastien") "m" _sb ("FATHER" _pxb)("MOTHER" _chb)
    ("BROTHER" _ab)("COUSIN" _cxb _psb)("AGE" 24))
(mp ("BarthËs") ("Emilie") "f" _eb ("FATHER" _pxb)("MOTHER" _chb)
    ("BROTHER" _ab _sb)("COUSIN" _cxb _psb)("AGE" 21))
(mp ("BarthËs") ("Papy") "m" _apb ("SON" _jpb _pxb)("AGE" 85))
(mp ("BarthËs") ("Mamy") "f" _mlb ("SON" _jpb _pxb)("HUSBAND" _apb)("AGE" 81))
(mp ("Labrousse" "BarthËs") ("Marie-GeneviËve") "f" _mgl ("FATHER" _apb)
    ("MOTHER" _mlb)("AGE" 48)("BROTHER" _jpb _pxb))
(mp ("Labrousse") ("Michel") "m" _ml ("WIFE" _mgl)("AGE" 52))
(mp ("Labrousse") ("Sylvie") "f" _sl ("FATHER" _ml)("MOTHER" _mgl)
    ("COUSIN" _psb _cxb _ab _sb _eb)("AGE" 19))
(mp ("Labrousse") ("Claire") "f" _cl ("FATHER" _ml)("MOTHER" _mgl)("SISTER" _sl)
    ("COUSIN" _psb _cxb _ab _sb _eb)("AGE" 18))
(mp ("Labrousse") ("AgnËs") "f" _al ("FATHER" _ml)("MOTHER" _mgl)
    ("SISTER" _sl _cl)("COUSIN" _psb _cxb _ab _sb _eb)("AGE" 15))
(mp ("Canac") ("Bruno") "m" _bc)

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

(m-definstance ORGANISM ("ABBREVIATION" "UTC")
               ("NAME" "UniversitÈ de Technologie de CompiËgne")
               ("EMPLOYEE" _jpb _dbb _df _gk _pt)
               ("STUDENT" _wms _hda))
(m-definstance ORGANISM  ("ABBREVIATION" "IC")
               ("NAME" "Imperial College")
               ("STUDENT" _psb))

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

```
;(setq *persons* (send *qh* '=process-query '(person)))
(defParameter *persons*
  (list _jpb _dbb _cxb _psb _pxb _chb _ab _sb _eb _apb _mlb _mgl _ml _sl _cl _al _wms
        _hda _es _ym _lv _df _cml _gk _jlg _pt))
(defVar *rule-set* nil)
(setq *rule-set* nil)
(defMacro ar (&rest rule)
  `(push ',rule *rule-set*))
;;;  Rules to close the family links
;;;
;;; If person P1 has-sex M
;;;              has-brother P2
;;;    then person P2 has-brother P1
(ar "m" has-brother has-brother)
;;;
;;; If person P1 has-sex M
;;;              has-sister P2
;;;    then person P2 has-brother P1
(ar "m" has-sister has-brother)
;;;
;;; If person P1 has-sex M
;;;              has-father P2
;;;    then person P2 has-son P1
(ar "m" has-father has-son)
;;;
;;; If person P1 has-sex M
;;;              has-mother P2
;;;    then person P2 has-son P1
(ar "m" has-mother has-son)
;;;
;;; If person P1 has-cousin P2
;;;    then person P2 has-cousin P1
(ar "m" has-cousin has-cousin)
(ar "f" has-cousin has-cousin)
;;;
;;; If person P1 has-sex M
;;;              has-son P2
;;;    then person P2 has-father P1
(ar "m" has-son has-father)
;;;
;;; If person P1 has-sex M
;;;              has-daughter P2
;;;    then person P2 has-father P1
(ar "m" has-daughter has-father)
;;;
;;; If person P1 has-sex M
;;;              has-wife P2
;;;    then person P2 has-husband P1
(ar "m" has-wife has-husband)
;;;
;;; If person P1 has-sex F
```

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


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

(defun apply-rule (p1 p2 rule)
  "produces a list of messages to send to modify the data.
Arguments:
   p1: person 1
   p2: person 2
   rule: (sex) rule to be applied
Return:
   nil or message to be sent."
  ;; note that we use =get and not =get-id that would require the local id of
  ;; the specified property, e.g. $t-person-name or $t-student-name, which
  ;; cannot be guarantied in a global rule
  (let ((sex (car rule))
        (sp-name (cadr rule))
```

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

#|
(defun apply-rule (p1 p2 rule)
  "produces a list of messages to send to modify the data.
Arguments:
   p1: person 1
   p2: person 2
   rule: (sex) rule to be applied
Return:
   nil or message to be sent."
  ;; note that we use =get and not =get-id that would require the local id of
  ;; the specified property, e.g. $t-person-name or $t-student-name, which
  ;; cannot be guarantied in a global rule
  (let ((sex (car rule))
        (sp-name (cadr rule))
        (ret-sp-name (caddr rule)))
    (if (and (equal sex (car (send P1 '=get 'HAS-SEX)))
             (member P2 (send P1 '=get sp-name)))
      `(send ',P2 '=add-sp ',ret-sp-name ',P1))))
|#

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

(defun close-family-links (family-list rule-set)
  "Function that takes a list of persons and computes all the links that must be added
to such persons to close the parent links brother, sister, father, mother, son,
daughter, cousin. It executes a loop until no more changes can occur, applying
rules from a rule set. The result is a list of instructions to be executed."
  (let ((change-flag t) action action-list (count 0))
    (print count)
    ;; Loop until no more rule applies
    (loop
      (unless change-flag (return nil))
      ;; reset flag
      (setq change-flag nil)
      ;; loop on couple of objects in the family-list
      (dolist (P1 family-list)
        (dolist (P2 family-list)
          ;; loop on each rule
          (dolist (rule rule-set)
```

```
            ;; if the rule applies it produces an instruction to be executed
            (setq action (apply-rule P1 P2 rule))
            ;; which is appended to the instruction list
            (when (and action
                       (not (member action  action-list :test #'equal)))
              ;; print a mark into the debug trace to tell user we are doing
              ;; some work
              (prin1 '*)
              (incf count)
              (if (eql (mod count 50) 0) (print count)) ; new line after 50 *s
              (push action action-list)
              ;; in addition we mark that a changed occured by setting the change flag
              (setq change-flag t))))))
    ;; return list of actions
    (reverse action-list)))

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

(defVar instructions)
(setq instructions (close-family-links *persons* *rule-set*))
(mapcar #'eval instructions)

;;; Adding orphans

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

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