Operational Concept Modelling Language

Dave Lambert and Enrico Motta

Table of Contents

1 Introduction

1.1 Provenance

This manual is substantially based on chapter 4 and appendix 1 of [Reusable Components for Knowledge Modelling], page 27 by Enrico Motta. Dave Lambert reformatted it as Texinfo, reorganized the material to something more appropriate to a user manual, and added the installation and namespace sections.

2 Installation

OCML is implemented in Lisp, and currently runs under Lispworks and SBCL.

2.1 Obtaining OCML

You can obtain the development version of OCML using Git[1].

```
git clone http://kmi-forge.open.ac.uk/git/ocml.git
```

This should leave you with a directory called ‘`ocml`’.

2.2 Setup

OCML uses the ASDF system definition package. You can find out more about ASDF at http://www.cliki.net/asdf. If you have ASDF installed, make sure the ‘`ocml.asd`’ file is accessible from `asdf:*central-registry*`, and then load it:

```
CL-USER > (asdf:operate ’asdf:load-op :ocml)
```

or, if you’re using a Lisp which integrates ASDF with `require`, just do:

```
CL-USER> (require :ocml)
```

You then need to initialize OCML:

```
CL-USER> (ocml:initialize-ocml)
```

2.3 Library locations

OCML finds its libraries (the ontologies) via logical pathnames. The logical host `ocml` holds the translations. Immediately after loading OCML, this will be set to include the basic OCML library:

```
CL-USER> (logical-pathname-translations "ocml")
(("ocml:library;basic;**;*" "/home/djl/dev/ocml/library/basic/"))
```

Additional translations can be added, for instance in the IRS, the translations look like this:

```
CL-USER> (logical-pathname-translations "ocml")
(("ocml:library;basic;**;*" "/home/djl/dev/ocml/documentation//library/basic/")
 ("ocml:library;**;*" "/home/djl/dev/irs/live/code/ocml/library/v5-0/**/*"))
```

Translations earlier in the list take precedence.

[1] http://git.or.cz/

3 Domain Modelling in OCML

3.1 Overview

OCML provides mechanisms for defining relations, functions, classes, instances, rules and procedures.

OCML supports the specification of three types of constructs: functional and control terms, and logical expressions.

3.1.1 Functional terms

A functional term - in short, a term - specifies an object in the current domain of investigation. A functional term can be a constant, a variable, a string, a function application or can be constructed by means of a special term constructor. This can be one of the following: if, cond, the, setofall, findall, quote and in-environment7 - see appendix 1 for a description of the semantics of these terms. Variables are represented as Lisp symbols beginning with a question mark, e.g. ?x is a variable. Strings are sequences of characters enclosed in double quotes, e.g. "string". A function application is a term such as `(fun {fun-term}*)`, where fun is the name of a function and fun-term a functional term. Functions are defined by means of the Lisp macro def-function, which is described in Section 3.6 [Functions], page 9.

Functional terms are evaluated by means of the OCML function interpreter, which is described in detail in appendix 1.

3.1.2 Control terms

Modelling problem solving behaviour involves more than making statements and describing entities in the world. Control terms are needed to specify actions and describe the order in which these are executed. OCML supports the specification of sequential, iterative and conditional control structures by means of a number of control term constructors, such as repeat, loop, do, if, and cond8. A Lisp macro, def-procedure, makes it possible to label parametrized control terms - i.e. to define procedures. Control terms are evaluated by means of a control interpreter. This is described in appendix 1.

3.1.3 Logical expressions

OCML also provides the usual machinery for specifying logical expressions. The simplest kind of logical expression is a relation expression, which has the form `(rel {fun-term}*)`, where rel is the name of a relation and fun-term is a functional term. More complex expressions can be constructed by using the logical operators `and`, `or`, `not`, `=>`, `<=>` and quantifiers - forall and exists. Operational semantics is provided for all operators and quantifiers. Relations are defined by means of the Lisp macro `def-relation`.

3.2 Ontologies

OCML organises its knowledge models using ontologies. When an ontology is defined, say ‘`O`’, it is possible to specify which ontologies are included in ‘`O`’ and as a result ‘`O`’ will include all the definitions from its parent ontologies. When conflicts are detected (e.g., the same concept is defined in two different parent ontologies), a warning is issued. Primitives for loading and selecting ontologies are also provided.

By default all ontologies are built on top of the OCML base ontology. This comprises twelve sub-ontologies which include the basic definitions required to reason about basic data types (e.g. lists, numbers, sets and strings), the OCML system itself and the OCML frame representation. Specifically, the following sub-ontologies are provided:

Meta	This ontology defines the concepts required to describe the OCML language. It includes constructs such as ‘OCML expression’, ‘functional term’, ‘rule’, ‘relation’, ‘function’, ‘assertion’, etc. This ontology is particularly important to construct reasoning components which can verify OCML models.
Functions	Defines the concepts associated with functions - e.g., it includes relations such as domain, range, unary-function, binary-function, etc.
Relations	Defines the various notions associated with relations. These include the universe and the extension of a relation, the definition of reflexive and transitive relations, partial and total orders, etc.
Sets	This ontology defines the notions associated with sets - e.g., ‘empty set’, ‘union’, ‘intersection’, ‘set partition’, ‘set cardinality’, etc.
Numbers	Defines the various concepts and operations required for reasoning about numbers and for performing calculations.
Lists	Defines the concepts and operations associated with lists. It includes classes such as list and atom; functions such as first, rest and append; and relations such as member.
Strings	Specifies the concepts and operations associated with strings - e.g., string, string-append, etc.
Mapping	This ontology defines the concepts associated with the mapping mechanism described earlier. It includes only three definitions: relation maps-to and functions meta-reference and domain-reference. The former takes a domain-level instance and returns the associated task/method level instance. The latter performs the inverse function.
Frames	Defines the concepts associated with the frame-based representation used in OCML. It comprise (meta-) classes such as class and instance; functions such as direct-instances and all-slot-values; relations such as has-one and has-at-most; and procedures such as append-slot-value.
Inferences	The purpose of this ontology is to provide a repository for defining functions and relations supporting the specification of KADS-like inferences. So far only a few such inferences have been added to this ontology to support different types of selection and sorting.
Environment	This ontology provides a kind of ‘environmental support’ for the construction of OCML models. It includes special operators like exec, which makes it possible to invoke procedures from rules, and procedures such as output, which prints out a message.

Task-Method
: This ontology provides the concepts required to model tasks and problem solving methods, i.e. to support the construction of task and problem solving models.

This set of ontologies provides a rich modelling platform from which to build other ontologies and/or problem solving models. It is natural to compare the OCML and Ontolingua base ontologies (Farquhar et al., 1996). There are two aspects which distinguish these two sets of ontologies: their nature and their scope.

The first difference is related to the operational nature of OCML. The Ontolingua base ontology is not concerned with operationality and therefore includes many non-operational definitions. The OCML base ontology is concerned with providing support for the construction of operational models. As a result it attempts to minimize the number of non-executable specifications. A typical approach is to weaken a non-operational definition to make it executable. For instance let's consider the function universe. In the Ontolingua base ontology the universe of a relation is defined as the set of all objects for which the relation is true. Of course this is not an operational definition. However, we can provide a weaker version of universe, called known-universe, which returns the set of all entities which are part of a tuple satisfying the relation in question. This function can be either defined separately from universe or attached to it to provide an operational definition.

The second difference concerns the scope of the two base ontologies. Both the Ontolingua and OCML base ontologies provide a rich set of definitions for domain modelling. In order to comply with the requirements imposed by the TMDA framework, the OCML base ontology provides support also for specifying tasks and problem solving methods.

3.3 Classes

OCML also supports the specification of classes and instances and the inheritance of slots and values through isa hierarchies.

Classes are defined by means of a Lisp macro, `def-class`, which takes as arguments the name of the class, a list (possibly empty) of superclasses, optional documentation, and a list of slot specifications, as illustrated by the definitions in the next box. These show a number of classes taken from the domain model for the Sisyphus-I office allocation problem (Linster, 1994; Chapter 9). This problem consists of allocating members of the YQT laboratory to the appropriate offices, according to a number of constraints.

```
(def-class YQT-member ()
  ((has-project :type project)
   (smoker :type boolean :cardinality 1)
   (hacker :type boolean :cardinality 1)
   (works-with :type YQT-member)
   (belongs-to-group :type research-group :value yqt)))

(def-class researcher (YQT-member))

(def-class secretary (YQT-member))

(def-class manager (YQT-member))
```

OCML provides support for the usual slot specification machinery which is found in frame-based languages. Specifically, it provides the following slot options.

`value`	A value which is inherited by all instances of a class.
`default-value`	A value which is inherited by all instances of a class, unless overridden by other values.
`type`	The value of this option should be a class, say C. This option specifies that all values of the associated slot should be instances of C. It is also possible to specify that a slot value must belong to one or more classes, say C1,...., Cn, by using the notation (or C1 C2..Cn).
`max-cardinality`	The maximum numbers of slot values allowed for a slot.
`min-cardinality`	The minimum numbers of slot values required for a slot.
`cardinality`	The numbers of slot values required for a slot. This option subsumes both `:min-cardinality` and `:max-cardinality`.
`documentation`	The value of this option is a string providing documentation for a slot.
`inheritance`	The inheritance mechanism used for dealing with default values. If `:merge` is used, then all default values inherited from different ancestors are collected. If `:supersede` is used, then default values inherited from more specific ancestors override those inherited from more generic ones.

3.4 Instances

Instances are simply members of a class. An instance is defined by means of def-instance, which takes as arguments the name of the instance, the parent of the instance (i.e. the most specific class the instance belongs to), optional documentation and a number of slot-value pairs. An example of instance definition, taken from the Sisyphus-I domain model, is shown in the box below.

```
(def-instance harry_c researcher
  ((has-project babylon)
   (smoker no)
   (hacker yes)
   (works-with jurgen_l thomas_d)))
```

In particular the above definition shows that a slot can have multiple values. In this case `harry_c` works both with `jurgen_l` and `thomas_d`.

3.5 Relations

Relations allow the OCML user to define labelled n-ary relationships between OCML entities. Relations are defined by means of a Lisp macro, `def-relation`, which takes as arguments the name of a relation, its argument schema, optional documentation and a number of relation options. An argument schema is a list (possibly empty) of variables. Relation options play two roles, one related to the formal semantics of a relation, the other to the operational nature of OCML. These roles are discussed in the next two sections.

3.5.1 Relation specification options

From a formal semantics point of view the purpose of a relation option is to help characterize the extension of a relation. The table below shows the relation options which can be used to provide formal relation specifications and, for each option, informally describes its semantics. A formal semantics to these options can be given in terms of the homonymous Ontolingua constructs.

Relation option	**Role in specification**
`:iff-def`	Specifies both sufficient and necessary conditions for the relation to hold for a given set of arguments.
`:sufficient`	Specifies a sufficient condition for the relation to hold for a given set of arguments.
`:constraint`	Specifies an expression which follows from the definition of the relation and must be true for each instance of the relation.
`:def`	This is for compatibility with Ontolingua: it specifies a constraint which is also meant to provide a partial definition of a relation.
`:axiom-def`	A statement which mentions the relation to which it is associated. It provides a mechanism to associate theory axioms with specific relations.

3.5.2 Operationally-relevant relation options

Relation options also play an operational role. Specifically, some relation options support constraint checking over relation instances while others provide proof mechanisms which can be used to find out whether or not a relation holds for some arguments. The table below lists the relation options which are meaningful from an operational point of view and informally describes their relevance to constraint checking and theorem proving.

Relation option	**Supports constraint checking**	**Provides proof mechanism**
`:sufficient`	No	Yes
`:prove-by`	No	Yes
`:lisp-fun`	No	Yes
`:iff-def`	Yes	Yes
`:constraint`	Yes	No
`:def`	Yes	No

As shown in the table, constraint checking is supported by the following keywords: `:constraint`, `:def` and `:iff-def`. While these have different model-theoretic semantics, from a constraint checking point of view they are equivalent. They all specify an expression which has to be satisfied by each known instance of the relevant relation.

The relation options `:iff-def`, `:sufficient`, `:prove-by` and `:lisp-fun` provide mechanisms for verifying whether or not a relation holds for some arguments. The first two—`:iff-def` and `:sufficient`—also play a specification role. The others—`:prove-by` and `:lisp-fun`—only play an operational role.

Both `:iff-def` and `:sufficient` indicate logical expressions which can be used to prove whether some tuple is an instance of a relation. From a theorem proving point of view there is an important difference between them. Let's suppose we are trying to prove that a tuple, say T, satisfies a relation, say R. If a `:sufficient` condition is tried and failed, the OCML proof system will then search for alternative ways of proving that T satisfies R. If an `:iff-def` condition is tried and failed, then no alternative proof mechanism will be attempted.

The relation options `:prove-by` and `:lisp-fun` are meant to support rapid prototyping and early validation by providing efficient mechanisms for checking whether a tuple satisfies a relation. The difference between :prove-by and :lisp-fun has to do with the expressions which are used as values to the two options: `:prove-by` points to a logical expression, `:lisp-fun` to a non-logical one (specifically a Lisp expression).

The box below provides an example of how the various types of relation options can be used concurrently to specify a relation and to support constraint checking and efficient proofs. The relation has-value, shown below, associates a design parameter to its value in a design model. The definition specifies that `?v` is the value of a parameter, `?p`, in a design model, `?dm`, if and only if the pair `(?p . ?v)` is an element of `?dm`. In addition, it also specifies the constraint that the value of a parameter has to be a member of its value range, if this has been specified. Finally, the definition includes a `:prove-by` option whose value is an expression which can be used for verifying whether the relation is satisfied for a triple, `(?p ?v ?dm)`. This expression provides an efficient proof method (by weakening the :iff-def statement which formally defines relation `has-value`) but does not contribute to the specification.

```
(def-relation HAS-VALUE (?p ?v ?dm)
  "Parameters have values w.r.t a particular design model"
  :iff-def (and (parameter ?p)
                (design-model ?dm)
                (element-of (?p . ?v) ?dm))
  :constraint (or (and (exists ?vr
                               (has-value-range ?p ?vr))
                       (element-of ?v ?vr))
                  (not (exists ?vr
                               (has-value-range ?p ?vr))))
  :prove-by (element-of (?p . ?v) ?dm))
```

A meta-option for non-operational specifications

As shown above, OCML provides a number of relation options, which play both a specification and an operational role. However, in some cases we might want to use a keyword only for specification and not operationally, for instance when we know that the value of the keyword in question is a non-operational expression. To cater for these situations, OCML provides a special meta-keyword, `:no-op`, which can be used to indicate that the enclosed relation option only plays a specification role. An example of its use is shown by the definition of relation range, which is shown below. In the example the keyword `:no-op` is used to indicate that the `:iff-def` specification of the relation is not operational - in particular it is not normally feasible to test the range of a function on all its possible inputs!

```
(def-relation RANGE (?f-r ?relation)
  "The range of a function or a binary relation is a relation which is
   true for any possible output of the function or second argument of
   the binary relation"
  :no-op (:iff-def (or
                   (and (function ?f-r)
                         (forall (?args ?result)
                                 (=> (= (apply ?f-r ?args) ?result)
                                     (holds ?relation ?result))))
                    (and (binary-relation ?f-r)
                         (forall (?x ?y)
                                 (=> (holds ?f-r ?x ?y))
                                     (holds ?relation ?y))))))
```

In the above definition it is worth highlighting the use of the special meta-relation holds. An expression such as `(holds <r> <arg1>....<argn>)` is satisfied if and only if the expression `(<r> <arg1>....<argn>)` is satisfied. Thus holds has variable arity: it can take one or more arguments. In particular the number of additional arguments in a holds statement reflects the arity of the relation passed as first argument. The relation holds has a 'special status' because it is the only relation with variable arity supported by OCML.

3.5.3 Summing up

The set of relation options discussed here aims to provide a flexible and versatile range of modelling constructs supporting various styles of modelling. While the emphasis is on operational modelling, OCML also supports formal specification. Moreover, it provides facilities for integrating a specification with efficient proof mechanisms.

3.6 Functions

A function defines a mapping between a list of input arguments and its output argument. Formally functions can be characterized as a special class of relations, as in KIF (Genesereth and Fikes, 1992). However, in operational terms there is a significant difference between a function and a relation: functions are applied to ground terms to generate function values; relation expressions can be asserted or queried. Thus, in accordance with the operational nature of OCML, functions are distinguished from relations.

Functions are defined by means of a Lisp macro, def-function. This takes as argument the name of a function, its argument list, an optional variable indicating the output argument (as in Ontolingua this is preceded by an arrow, `->`), optional documentation and zero or more function specification options. These are `:def`, `:constraint`, `:body` and `:lisp-fun`.

The option :constraint provides a way to constrain the domain (i.e. the set of possible inputs) of a function. It specifies a logical expression which must be satisfied by the input arguments of the function. The `:def` option indicates a logical expression which 'defines' the function. This expression should be predicated over both (some) input arguments and the output variable. Operationally, the expression denoted by the `:constraint` option provides a mechanism for testing the feasibility of applying a function to a set of arguments. The expression denoted by :def provides a mechanism for verifying that the output produced by a function application is consistent with the formal definition of the function.

Finally, the options `:body` and `:lisp-fun` provide effective mechanisms for computing the value of a function. The former specify a functional term which is evaluated in an environment in which the variables in the function schema are bound to the actual arguments. The latter makes it possible to evaluate an OCML function by means of a procedural attachment, expressed as a Lisp function. The arity of this Lisp function should be the same as that of the associated OCML function.

```
(def-function filter (?l ?rel) -> ?sub-l
  "Returns all the elements in ?l which satisfy ?rel"
```

```
    :def (and (unary-relation ?rel)
              (list ?l))
              (list ?sub-l)
              (=> (and (member ?x ?sub-l)
                       (holds ?rel ?x))
                  (member ?x ?l)))
    :body (if (null ?l)
              ?l
              (if (holds ?rel (first ?l))
                (cons (first ?l)
                      (filter (rest ?l) ?rel))
                (filter (rest ?l) ?rel))))
```

The above definition shows an example of the use of `def-function`. The OCML function filter takes as arguments a list, `?l`, and a unary relation, `?rel`, and returns the elements of `?l` which satisfy `?rel`. As illustrated by the definition, the `:def` option provides a declarative way of specifying a function; the option `:body` an effective way of computing its value, for a given set of input arguments.

3.7 Procedures

Procedures define actions or sequences of actions which cannot be characterized as functions between input and output arguments. For example, the procedure below defines the sequence of actions needed to set the value of a slot. These include a unassert statement, which removes any existing value from the slot, and a tell statement, which adds the new value. Both tell and unassert are procedures. The former takes a ground logical expression and adds it to the current model. The latter takes a relation expression and removes from the current model all assertions which match it. Note that in accordance with the uniform view of a knowledge model, slot changes are carried out by means of generic assertion and deletion operations (i.e. in terms of tell and unassert).

```
  (def-procedure SET-SLOT-VALUE (?i ?s ?v)
    :constraint (and (instance-of ?i ?c)
                     (slot-of ?s ?c))
    :body (do
            (unassert (list-of ?s ?i ?any))
            (tell (list-of ?s ?i ?v))))
```

3.8 Namespaces

OCML uses Lisp symbols to name things. This is convenient, but awkward when dealing with ontologies on a global scale. OCML now supports a convenient interface for namespaces which is very similar to the XML namespaces scheme. Three things are involved in this extension:

1. OCML ontologies must indicate the URI which identifies them.
2. Namespaces prefixes are registered with OCML before use.
3. Namespaced symbols are prefixed with `#_`.

Ontology definitions should include a `:namespace-uri` initialization argument, which is a string holding the URI identifying the ontology. If you do not supply a namespace URI, OCML will assign one based on the ontology's name.

```
  (def-ontology colours
      :author "dave"
```

```
    :namespace-uri "http://www.kmi.open.ac.uk/colours#"
    :files ())

(def-ontology fruits
    :author "dave"
    :namespace-uri "http://www.kmi.open.ac.uk/fruits#"
    :files ())
```

Next, we tell OCML which prefixes we intend to use:

```
(register-namespace "colours" 'colours)
(register-namespace "fruit" 'fruits)
```

We can then use these prefixes in ontologies, using a Lisp reader macro, `#_`. Classes, instances, and relations can all be named in this way. Viz:

```
(in-ontology colours)

(def-class #_colour ())

(def-class #_blue (#_colour)

(def-class #_red (#_colour)

(def-class #_orange (#_colour)

(in-ontology fruit)

(def-class #_fruit ())

(def-class #_orange (#_fruit)
 (#_colors:has-colour #_colours:orange))
```

Here, `#_colours:orange` and `#_fruit:orange` are different symbols, from different namespaces and ontologies, used together in one ontology.

When Lisp reads a namespaced symbol, it converts it to a symbol by concatenating the value of the referenced ontology's `namespace-uri` slot with the symbol. Thus, `#_fruit:orange` becomes `OCML::|http://www.kmi.open.ac.uk/fruits#orange|`. It is important to remember that the `#_` syntax is merely a shorthand for the long, quoted Lisp symbol. If you so wish, you can enter the long form directly, as a normal Lisp symbol. Given this, namespaced variables interoperate fully with non-namespaced ones. The prefixes themselves are discarded after they have been read in by the Lisp reader. Namespaced symbols are case sensitive: `#_foo` is distinct from `#_Foo`.

Further examples from a Lisp listener session:

```
OCML> (in-ontology fruits)
#<OCML-ONTOLOGY FRUITS>
OCML> '#_colours:orange
OCML::|http://www.kmi.open.ac.uk/fruits#orange|
OCML> '#_orange
OCML::|http://www.kmi.open.ac.uk/fruits#orange|
OCML> (register-namespace "colors" 'colours)
#<OCML-ONTOLOGY COLOURS>
OCML> (in-ontology colours)
#<OCML-ONTOLOGY COLOURS>
OCML> '#_orange
OCML::|http://www.kmi.open.ac.uk/colours#orange|
OCML> (eq '#_colours:orange '#_colours:orange)
T
```

```
OCML> (eq '#_colors:orange '#_colours:orange)
T
OCML> (eq '#_colors:orange 'ocml::|http://www.kmi.open.ac.uk/colours#orange|)
T
OCML> (eq '#_colours:orange '#_colours:Orange)
NIL
```

The notation `#_` is modelled after the WSML syntax for IRIs, `_"http://www.deri.org/foo"`.

3.9 Translation between representation languages

While OCML is an excellent language for knowledge modelling, poor taste is widespread and we must often translate to and from other representations. For this, we use the `translate` generic function. `translate` takes 4 required arguments: keywords indicating source and destination lanuages, an object in the source language, and a stream to write the output to.

```
(translate :ocml :rdf 'colours *standard-output*)
```

XXX At present, not all translations are available through this interface, and the methods of returning the result are idiosyncratic.

3.10 The ask-tell interface

In particular, in the case of OCML, this generic idea of providing a uniform level of description to a hybrid formalism has been instantiated by providing a Tell-Ask interface (Levesque, 1984), which use logical expressions (i.e. a relation-oriented view) when modifying or querying an OCML model, independently of whether the query in question concerns a class, a slot, or a 'ordinary' relation. The key to this integrated view is the fact that classes and slots are themselves relations; classes are unary relations and slots are binary ones. In addition to supporting a generic Tell-Ask interface, this property makes it possible to provide 'rich' specifications of classes and slots. In particular, because these are relations, it is possible to characterize them by means of the relation options discussed in section Section 3.5 [Relations], page 6. For instance, the definition below specifies the class of empty sets in terms of an `:iff-def` relation option.

```
(def-class EMPTY-SET (set) ?set
   :iff-def (not (exists ?x (element-of ?x ?set))))
```

The generic Tell-Ask interface Tell: a generic assertion-making primitive

OCML provides a generic assertion-making primitive, tell, which provides a uniform mechanism for asserting facts, independently of whether these refer to slot-filling assertions, new class instances, or simply relation instances. For example I can use tell to add a new value to the list of projects carried out by harry_c as follows.

```
? (tell (has-project harry_c mlt))
(HAS-PROJECT HARRY_C MLT)
```

Analogously I can add a new instance of class researcher simply by stating:

```
? (tell (researcher mickey_m))
(RESEARCHER MICKEY_M)
```

The relation-centred view makes it possible to examine the contents of an OCML model simply by asking whether a logical statement is satisfied. The OCML proof system will then carry out the relevant inference and retrieval operations, depending on whether the relation being queried is a slot, a class, or an 'ordinary' relation. The process is however transparent to the user. For instance, I can find out about the projects in which harry_c is involved - after the assertion shown above these are now babylon and mlt - by using the Lisp macro ask to pose the query (has-project harry_c ?c). The resulting interaction with the OCML proof system is shown below.

```
? (ask (has-project harry_c ?c))
Solution: ((HAS-PROJECT HARRY_C BABYLON))
More solutions?  (y or n)  y
Solution: ((HAS-PROJECT HARRY_C MLT))
More solutions?  (y or n)  y
No more solutions
```

This uniform, relation-centred view over OCML models also provides a way to index inferences. For instance, when answering the above query, the OCML proof system will first retrieve and order all inference mechanisms applicable to a query of type has-project - e.g., these might include assertions of type has-project, relation options associated with has-project and the relevant backward rules13 - and will then try these in sequence, to generate one or more solutions to the query.

4 Rule-based reasoning

OCML can reason with backward and forward chaining rules.

4.1 Backward-chaining rules

A backward rule consists of a number of backward clauses, each of which is defined according to the following syntax:

```
backward-clause ::= (relation-expression {if {logical-expression}+})
```

Each backward clause specifies a different goal-subgoal decomposition. When carrying out a proof by means of a backward rule the OCML interpreter will try to prove the relevant goal by firing the clauses in the order in which these are listed in the rule definition. As in Prolog, depth-first search with chronological backtracking is used to control the proof process.

Both semantically and operationally a backward chaining rule is the same as a :sufficient relation option: they both provide an expression which is sufficient to verify that a tuple holds for a relation. Thus, one might wonder whether rules are needed at all. In practice the advantage of including backward rules in the language is that these provide a modular mechanism for refining existing (possibly generic) relation specifications, for instance in cases where application-specific knowledge is needed to complete the specification of a relation. To clarify this point let's consider an example taken from the KMI office allocation problem (see chapter 9 for an extensive analysis of this application).

The relation has-value-range is defined in the parametric design task ontology to characterize the set of possible values which can be assigned to a design parameter - see chapter 6. When building an application model the generic has-value-range specification is usually refined, so that the space of possible values that can be assigned to a particular parameter is precisely defined. A modular way to achieve this is to refine the definition of the relation by means of the appropriate backward chaining rules. The rules shown below fulfil this purpose for two of the classes of parameters present in the KMI office allocation domain: professors and secretaries. In this example the former can only go into a double room; the latter into a large room next to the entrance.

```
(def-rule has-value-range-1
  ((has-value-range-gen ?m ?l)
   if
   (professor (domain-reference ?m))
   (= ?l (setofall ?r (double-a-type-room ?r usable yes)))))

(def-rule has-value-range-2
  ((has-value-range-gen ?m ?l)
   if
   (secretary (domain-reference ?m))
   (= ?l (setofall ?r
                 (and (room ?r size ?n usable yes)
                      (> ?n 1)
                      (close-to ?r kmi-entrance))))))
```

4.2 Forward-chaining rules

OCML also allows the user to define forward rules. A forward rule comprises zero or more antecedents and one or more consequents. Antecedents are restricted to relation expressions, while any logical expression can be a consequent. When a forward rule is executed, OCML treats each consequent as a goal to be proven and attempts to prove them, until one fails. This mechanism makes it possible to integrate data-driven and goal-driven reasoning and to specify arbitrarily complex right hand sides.

A special operator, `exec`, is provided to allow OCML users to introduce control (and therefore functional) terms in the right hand side of a rule. In particular, two useful procedures are tell, to assert new facts, and output, to produce output. A simple example showing how to use these in a forward chaining rule is given below.

```
(def-rule has-project-cond
    (has-project ?x ?y)
  then
  (exec (tell (project-covered-by ?y ?x)))
  (exec (output "has-project-cond rule deduced (project-covered-by ~S ~S)~%" ?y ?x)))

(enable-fc-watcher-mode)

(def-instance larry-e researcher
  ((has-project tigris)
   (smoker no)
   (hacker yes)
   (works-with harry_d)))
```

This produces the output '`has-project-cond rule deduced (project-covered-by TIGRIS LARRY-E)`'.

While forward rules can be useful in a number of situations when building application models (e.g. to define watchers, which are triggered whenever some situation arises in a knowledge base), they are not essential to the model building process. The reason for this is that knowledge-level modelling is mainly about constructing definitions, while forward-chaining rules are about behaviour. Thus they can be used in place of procedures to describe behaviour but they cannot replace constructs for relation or function specification.

`run-fc-rules` [Function]
`run` [Function]

Explicitly run forward-chaining rules in the current ontology.

`enable-fc-watcher-mode` **&optional** *packets* [Function]
`disable-fc-watcher-mode` [Function]

Enable and disable the automatic triggering of forward chaining rules.

`trace-fc` [Function]
`untrace-fc` [Function]

Turn on and off the tracing of forward chaining rules.

5 Modelling issues

5.1 Object-oriented and relation-oriented approaches to modelling

When describing classes and instances I made use of standard object-oriented terminology and talked about slots having values and instances belonging to classes. This object-centred approach is in a sense orthogonal to the relation-centred one which I used when discussing relations and logical expressions. The former focuses on the entities populating a model and then associates properties to them; the latter centres on the type of relations which characterize a domain and then uses these to make statements about the world. These two approaches to modelling/representation have complementary strengths and weaknesses and for this reason they are often combined in knowledge representation and modelling languages, to provide hybrid formalisms (Fikes and Kehler, 1985; Yen et al., 1988).

In the context of a knowledge modelling language (rather than a knowledge representation one) the main advantage gained from combining multiple paradigms is one of flexibility. Both object-oriented and relation-oriented approaches provide conceptual frameworks which make it possible to impose a view over some domain. The choice between one or the other can be made for ideological or pragmatic reasons - e.g. whether the target delivery environment is a rule-based shell or an object-oriented programming environment. In the specific context of an operational modelling language, such as OCML, another benefit, which is gained by providing support for both object-oriented and relation-oriented modelling, is that these approaches are naturally associated with particular types of inferences. Object-orientation provides the structure for inheritance and automatic classification; relation-orientation is normally associated with constraint-based and rule-based reasoning.

While the integration of multiple paradigms provides the aforementioned benefits, when describing or interacting with a knowledge model, it is useful to abstract from the various modelling paradigms and inference mechanisms integrated in the model and characterize it at a uniform level of description. Specifically, in accordance with a view of knowledge as a competence-like notion (Newell, 1982), it is useful to decouple the level at which we describe what an agent knows from the level at which we describe how the agent organizes and infers knowledge. Such an approach is used - for instance - in the Cyc system (Lenat and Guha, 1990), which integrates multiple knowledge representation techniques (the heuristic level), but provides a uniform interface to the Cyc knowledge base (the epistemological level). A similar approach is also followed by Levesque (1984), which describes a logic-based query language which can be used to communicate with a knowledge base at a functional level, independently from the data and inference structures present in the knowledge base.

6 Functional View of OCML

A functional view of a knowledge representation system focuses on the services the system provides to the user (Levesque, 1984; Brachman et al., 1985). Basically, there are three kinds of services provided by OCML: i) operations for extending/modifying a model; ii) interpreters for functional and control terms; and iii) a proof system for answering queries about a model. Extensive details on the model extension/modification facilities provided by OCML were given in earlier sections. The interpreters and the proof system are described in detail in appendix 1.

6.1 Mapping

The constructs presented so far provide extensive support for domain modelling. In order to fully support the TMDA framework additional constructs are needed, for specifying tasks and methods and for mapping entities at the task/method level to entities at the domain level. As already said, the conceptual vocabulary required to model tasks and methods is not hardwired in the language but is defined as a specialized ontology. This will be presented in the next chapter; in the next sections I will discuss the two kinds of mapping constructs supported by OCML: relation mapping and instance mapping.

6.2 Instance mapping

Figure 3.8 XXX illustrated a simple example in which a class of concepts at the task/method level (parameter) is mapped to a class of concepts at the domain level (employee). This is a very common situation when developing systems through reuse: a problem solving, domain-independent model imposes a particular view over a set of domain concepts (Fensel and Straatman, 1996).

A simple case is one in which a domain view is constructed by direct association of task level concepts to domain level concepts. For instance, a parametric design view over the Sisyphus-I domain can be imposed simply by creating parameter instances and associating them to YQT members. Thus, the set of parameters and the set of YQT members are associated but kept distinct. This solution is appealing for two reasons: it supports reuse of modular components and does not confuse the two different types of concepts; parameters and YQT members maintain different sets of properties and different semantics, thus avoiding a situation in which a design parameter has a wife and a YQT member has a value range.

Instance mapping is supported in OCML by means of the Lisp macro def-upward-class-mapping. This takes the names of two classes as arguments and associates each instance of the first class to a purpose-built instance of the second class. By default the relation maps-to is used to associate the task level instance to the domain level one.

In the Sisyphus-I application model we should then state:

```
(def-upward-class-mapping yqt-member yqt-parameter)
```

The above form iterates over each instance of class yqt-member, say I, creating a new instance of class yqt-parameter, say Meta-I, and associating this to I, i.e., asserting (maps-to Meta-I I). Hence, if we now ask for the mapping between parameters and YQT members we get the following results.

```
? (ask (maps-to ?z ?x)t)

Solution: ((MAPS-TO YQT-PARAMETER-EVA_I EVA_I))
Solution: ((MAPS-TO YQT-PARAMETER-MONIKA_X MONIKA_X))
Solution: ((MAPS-TO YQT-PARAMETER-ULRIKE_U ULRIKE_U))
Solution: ((MAPS-TO YQT-PARAMETER-UWE_T UWE_T))
Solution: ((MAPS-TO YQT-PARAMETER-JOACHIM_I JOACHIM_I))
Solution: ((MAPS-TO YQT-PARAMETER-HANS_W HANS_W))
Solution: ((MAPS-TO YQT-PARAMETER-MICHAEL_T MICHAEL_T))
Solution: ((MAPS-TO YQT-PARAMETER-ANGY_W ANGY_W))
Solution: ((MAPS-TO YQT-PARAMETER-JURGEN_L JURGEN_L))
Solution: ((MAPS-TO YQT-PARAMETER-KATHARINA_N KATHARINA_N))
Solution: ((MAPS-TO YQT-PARAMETER-THOMAS_D THOMAS_D))
Solution: ((MAPS-TO YQT-PARAMETER-HARRY_C HARRY_C))
Solution: ((MAPS-TO YQT-PARAMETER-ANDY_L ANDY_L))
Solution: ((MAPS-TO YQT-PARAMETER-MARC_M MARC_M))
Solution: ((MAPS-TO YQT-PARAMETER-WERNER_L WERNER_L))
```

The advantage of this solution is that parameters and YQT members maintain their separate identities, as shown in the next box.

```
? (describe-instance 'YQT-PARAMETER-WERNER_L)

Instance YQT-PARAMETER-WERNER_L of class YQT-PARAMETER

HAS-VALUE-RANGE: (C5-123 C5-122 C5-121 C5-120 C5-119 C5-117 C5-116 C5-113 C5-114 C5-115)

? (describe-instance 'WERNER_L)

Instance WERNER_L of class RESEARCHER

HAS-PROJECT: RESPECT

SMOKER: NO

HACKER: YES

WORKS-WITH: ANGY_W, MARC_M

GROUP: YQT
```

Formally, a mapping can be characterized as an association between an object, say o, and its meta-object, m-o, so that the entity denoted by the entity denoted by m-o is the same as the entity denoted by o (Genesereth and Nilsson, 1988). This notion can be formalized in OCML by means of the following definition.

```
 (def-relation maps-to (?x ?y)
  "This relation allows the user to specify an association between
   an object at the task layer and one at the domain layer.
   Formally ?y denotes the object denoted by the object denoted by ?x"
  :no-op (:iff-def (= ?y (denotation ?x))))
```

6.3 Relation mapping

Instance mapping works only in those cases in which imposing a view over a domain can be reduced to creating task-level 'mirror images' for a finite number of domain-level objects. A more general scenario is one in which there is some relation defined at the task/method level which needs to be reflected to the domain level in a dynamic fashion. A well known example is that in which domain concepts or statements are viewed as hypotheses at the problem solving level. This association is typically dynamic, given that hypotheses are considered as such only for a particular time-slice of the problem solving process. These situations can be modelled in OCML by means of relation mappings.

A relation mapping provides a mechanism to associate rules and procedures to a relation, say R, so that when a query of type R is posed, or assertions of type R are made at the task/method level, these events can be reflected to the domain level. The purpose of these reflection actions is to ensure that the consistency between domain and task/method levels is maintained.

An example of an upward relation mapping is illustrated by the definition below, which is taken from an application model developed for the Sisyphus-I problem. The mapping is an upward one, in the sense that it is used to lift (van Harmelen and Balder, 1992) the office allocation statements existing at the domain level to the problem solving level. Specifically, the goal of this mapping is to associate the relation current-design-model, which is used by the parametric design problem solver to indicate the design model associated with the current design state, to the set of in-room assertions present in the current snapshot of the domain knowledge base. The relation maps-to is used to retrieve the parameter associated with each particular YQT member.

```
(def-relation-mapping current-design-model :up
   ((current-design-model ?dm)
    if
    (= ?dm (setofall (?p . ?v)
                     (and (in-room ?x ?v)
                          (maps-to ?p ?x))))))
```

An upward relation mapping ensures that when a task/method level relation is needed the relevant information is obtained from the domain level. Of course, problem solving is also about inferring knowledge and retracting previously held assertions. Hence, OCML also supports downward relation mappings. These divide into two categories, :add and :remove. The former specifies a procedure which is activated when a new relation instance is asserted. The latter specifies a procedure which is activated when a relation instance is removed. In the case of relation current-design-model relation mappings are needed to ensure that when the design model considered by the problem solver is modified, the relevant changes are reflected onto the domain model - see definition below.

```
(def-relation-mapping current-design-model (:down :add)
  (lambda (?x)
    (do
      (unassert (in-room ?any-m ?any-r))
      (loop for ?pair in ?x
            do
            (if (maps-to (first ?pair) ?z)
              (tell (in-room ?z (rest ?pair))))))))
```

Finally, the definition below shows the :remove downward mapping associated with relation current-design-model: it simply removes the domain level assertions associated with the design model which is passed as argument to the relation instance being retracted.

```
(def-relation-mapping current-design-model (:down :remove)
   (lambda (?x)
     (loop for ?pair in ?x
            do
            (if (maps-to (first ?pair) ?z)
              (unassert (in-room ?z (rest ?pair)))))))
```

7 Reference

7.1 Functional Term Constructors

7.1.1 setofall

`setofall` finds all solutions (i.e. environments) to basic-log-expression and then returns the list obtained by instantiating template in all the returned environments, ensuring that the list contains no duplicates. If no solutions are found then the empty list is returned.

7.1.2 findall

`findall` is the same as setofall except that it does not remove duplicate solutions.

7.1.3 the

the finds one solution (i.e. environment) to basic-log-expression and then returns the list obtained by instantiating template in the returned environment. If no solutions are found then the constant :nothing is returned.

7.1.4 in-environment

The primitive in-environment takes a list, possibly empty, of pairs ((var1 . term1)...) and a body, and returns the result of evaluating this in an environment in which each vari is bound to termi.

7.1.5 quote

The value of an expression such as (quote term) is term.

7.1.6 if

The first action which is carried out when evaluating an if-term is to check whether log-expression is satisfied. If this is the case, then then-term is evaluated in the environment which satisfies log-expression. If log-expression cannot be satisfied in the current model, then there are two possibilities. If else-term is specified, then this is evaluated, and the value obtained is returned as the value of the if-term. If else-term is not present and log-expression cannot be proved, then the constant :nothing is returned.

7.1.7 cond

The interpreter iterates through each clause of a cond-term, until it finds one whose log-expression is satisfied. If none is found, then :nothing is returned. Otherwise, let's assume cond-clausei. is the first clause whose log-expression is satisfied. In this case the value of the cond-term is obtained by evaluating the term associated with cond-clausei.

7.2 Control Term Constructors

7.2.1 in-environment

The primitive in-environment takes a list, possibly empty, of pairs ((var1 . term1)...) and a control-body, and returns the result of evaluating this in an environment in which each vari is bound to termi.

7.2.2 if

The first action which is carried out when evaluating an if-control-term is to check whether log-expression is satisfied. If this is the case, then then-control-term is evaluated in the environment which satisfies log-expression. If log-expression cannot be satisfied in the current model, then there are two possibilities. If else-control-term is specified, then this is evaluated, and the value obtained is returned as the value of the if-control-term. If else-control-term is not present and log-expression cannot be proved, then the constant :nothing is returned.

7.2.3 cond

The interpreter iterates through each clause of a cond-control-term, until it finds one whose log-expression is satisfied. If none is found, then :nothing is returned. Otherwise, let's assume cond-clausei. is the first clause whose log-expression is satisfied. In this case the value of the cond-control-term is obtained by evaluating the control-term associated with cond-clausei.

7.2.4 loop

The control construct loop provides a simple mechanism for iterating over lists. It first evaluates a term, which should return a list, say L. Then it iterates over each element of L, say I, and evaluates control-term in an environment in which variable is bound to I.

7.2.5 do

The control construct do is a simple sequencing primitive. The control terms in its body are evaluated sequentially, once only.

7.2.6 repeat

The control term constructor repeat repeats the control term(s) specified in its body until the end test is satisfied, if the test has the form 'until test'. Otherwise, if the test has the form 'while test', then repeat-actions stops as soon as the test fails. If the end test is specified after the control terms, then the control terms are carried out at least once - i.e. the end test is verified at the end of each cycle. If the end test is specified before the control terms, then the test is verified at the beginning of each cycle. If no test is provided, then all control expression in the body of a repeat-actions are repeated ad infinitum.

7.2.7 return

This is a simple way of exiting from the body of a loop or repeat construct. When a control term such as (return term) is encountered, the most specific loop or repeat construct in the current execution stack is exited and the value obtained from evaluating term is returned.

7.3 Grammar

```
loop-control-term ::= (loop for variable in term do {control-term}+)
do-control-term  ::= (do-actions {control-term}+)
repeat-control-term  ::= (repeat-actions {end-test} {control-term}+ ) |
                                (repeat-actions {control-term}+  {end-test})
cond-control-term ::= (cond {cond-control-clause}+)
setofall-term           ::= setofall template basic-log-expression
template          ::= nil | (term . term)
```

```
term                 ::= constant | variable | string | (fun {term}*) |
                                findall-term | the-term |
                                in-env-term | quote-term |
                                if-term | cond-term
fun                  ::= the name of a function or a term constructor
constant          ::= A symbol whose first character is not '?'
variable          ::= A symbol whose first character is '?'
string  ::= A lisp string, e.g. "string".
log-expression  ::= quant-log-expression | basic-log-expression
quant-log-expression  ::= (forall schema-or-var log-expression)|
(exists schema-or-var log-expression)
basic-log-expression ::= (and {log-expression}+) |
(or {log-expression}+) |
(=> log-expression log-expression) |
(<=> log-expression log-expression)
(not log-expression) |
rel-expression
schema-or-var ::= schema | variable
schema ::= (variable . schema) | nil
rel-expression  ::= (rel {term}*)
rel  ::= a symbol naming a relation
findall-term ::= findall template basic-log-expression
in-env-term  ::= in-environment pairs body
pairs ::= nil | (pair . pairs)
pair  ::= (variable . term)
body ::= term
quote-term ::= 'term | (quote term)
the-term ::= the template basic-log-expression
if-term ::= (if log-expression then-term {else-term})
then-term ::= term
else-term ::= term
cond-term ::= (cond {cond-clause}+)
cond-clause ::= (log-expression term)
in-env-control-term ::= in-environment pairs control-body
pairs ::= nil | (pair . pairs)
pair  ::= (variable . term)
control-body ::= control-term
control-term ::= term | in-env-control-term | if-control-term |
cond-control-term | do-control-term |
loop-control-term | repeat-control-term |
return-control-term
if-control-term ::= (if log-expression then-control-term {else-control-term})
then-control-term  ::= control-term
else-control-term  ::= control-term
cond-control-term ::= (cond {cond-control-clause}+)
cond-control-clause ::= (log-expression control-term)
loop-control-term ::= (loop for variable in term do
{control-term}+)
do-control-term  ::= (do-actions {control-term}+)
repeat-control-term  ::= (repeat-actions {end-test} {control-term}+ ) |
(repeat-actions {control-term}+  {end-test})
end-test ::= while test | until test
test ::= log-expression
return-control-term ::= (return term)
if-control-term ::= (if log-expression then-control-term
{else-control-term})
return-control-term ::= (return term)
```

7.4 Inheritance and Default Values

Generally speaking default values are values which apply unless other alternatives can be used. In the OCML language the notion of default value is operationalized as follows.

Instances inherit values and default values from their superclasses down the inheritance hierarchy specified by instance-of and subclass-of links. For a given slot, say s, of a sample instance, say `I`, the following scenarios can arise:

1. `I` has not inherited any default value. In this case the value of s in `I` is given by all the values `I` has inherited from its superclasses, plus any value locally specified for slot s of `I`.
2. `I` has inherited some default values as well as non-default ones. In this case the default values are ignored and rule (i) is applied. We say that the default values are overridden by the non-default ones.
3. `I` has inherited only default values and local values have been specified. As in the previous case, the default values are ignored and only the local values are considered.
4. `I` has inherited only default values and no local values have been specified. In this case there are two possibilities. If the :inheritance facet has not been specified, or it has been specified and it is :merge, all default values apply. If the :inheritance facet has been specified and it is :supersede, then the value of s in `I` is obtained by (i) ranking the ancestors of `I` according to the class precedence order of the parent of `I`, and (ii) retrieving the default value of the first class in the class precedence order which specifies a (default) value for s. The details of the algorithm used to compute the class precedence order are given at pp. 782-786 of the Common Lisp specification (Steele, 1992). This algorithm produces a total order (if this exists) based on two ordering principles: (i) a class, say C, precedes all its direct superclasses, and (ii) a direct superclass of C precedes the direct superclasses of C specified to its right in the list of direct superclasses of C.

8 Interpreters and Proof System

8.1 Functional term interpreter

The OCML interpreter is implemented by means of a Lisp macro, `ocml-eval`. This evaluates a functional term, term, in an environment, env, according to the following rules.

1. If term is a variable, then the binding of term in env is returned.
2. If term is a string or a constant, then term is returned.
3. If term has the format (pfun term0, ..., termn), with n <= 0, where pfun is a primitive term constructor, then term is evaluated in env, according to criteria which depend on pfun.
4. If term has the format (fun term0, ..., termn), with n <= 0, where fun is the name of a function, and a Lisp body associated with fun exists, then ocml-eval returns the value obtained by applying the Lisp body to the values obtained by evaluating each termi in env.
5. If term has the format (fun term0, ..., termn), with n <= 0, where fun is the name of a function, and no Lisp body associated with fun exists, then ocml-eval returns the value obtained by applying the body of fun to the values obtained by evaluating each termi in env.
6. In all other cases ocml-eval signals an error.

8.2 Control term interpreter

Control terms are interpreted in a manner analogous to functional terms. The control term interpreter is implemented by a Lisp macro, ocml-control-eval, which has the following behaviour.

1. If term is a functional term, then it is evaluated according to the rules given in @ref{Functional term interpreter}.
2. If term has the format (proc term0, ..., termn), with n <= 0, where proc is a primitive control operator, then term is evaluated in env, according to criteria which depend on proc.
3. If term has the format (proc term0, ..., termn), with n <= 0, where proc is the name of a procedure, and a Lisp body associated with proc exists, then ocml-control-eval returns the value obtained by applying the Lisp body to the values obtained by evaluating each termi in env.
4. If term has the format (proc term0, ..., termn), with n <= 0, where proc is the name of a procedure, and no Lisp body associated with proc exists, then ocml-control-eval returns the value obtained by applying the body of proc to the values obtained by evaluating each termi in env.
5. In all other cases ocml-control-eval signals an error.

8.3 Proof system

Procedure for proving basic goal expressions in OCML

Let's suppose we want to find all solutions to a basic goal expression, say G, with format (rel {fun-term}*), where rel is the name of a relation and fun-term a functional term. In general we might be interested in one, some or all solutions. Therefore the order in which solutions are generated might be important. The algorithm used by the OCML proof system is as follows.

1. If rel is not a defined relation, then signal an error. Otherwise initialize SOL1, SOL2, SOL3, SOL4, SOL5 and SOL6 to the empty set and go to step 2.
2. Retrieve all the assertions present in the current model, whose type (i.e. first element) is rel. Match each assertion with G. All successful matches, we call this set SOL1, provide solutions to G. Go to step 3.
3. If a Lisp attachment exists for rel, then evaluate it in the Lisp environment to find eventual additional solutions to G, say SOL2. Go to step 8.
4. If a :prove-by proof condition, say prove-rel-expression, has been specified for relation rel, then compute all solutions to prove-rel-expression, say SOL3. Each of these is also a solution to G. Go to step 8.
5. If a :iff-def proof condition, say iff-rel-expression, has been specified for relation rel, then compute all solutions to iff-rel-expression, say SOL4. Each of these is also a solution to G. Go to step 8.
6. If a :sufficient proof condition, say suff-rel-expression, has been specified for relation rel, then compute all solutions to suff-rel-expression, say SOL5. Each of these is also a solution to G. Go to step 7.
7. If a backward chaining rule has been specified, associated with relation rel, then invoke it to find all other solutions to G, say SOL6. Go to step 8.
8. The set of all solutions to query G is obtained by appending the lists SOL1, SOL2, SOL3, SOL4, SOL5 and SOL6.

The algorithm shown above provides an operational semantics for the various relation-forming constructs provided by OCML. In particular the following two points should be highlighted.

Assertions inherited through an isa hierarchy are always cached at definition time. This means that they are retrieved at step 2, when the goal is matched against the current set of known facts.

The results returned by non-logical mechanisms such as Lisp attachments and :prove-by are only merged with the results obtained by simple assertion-matching (step 2). In other words they are meant to provide efficient proof mechanisms which override those provided by definition-forming options, such as :iff-def and :sufficient.

8.4 Proof rules for non-basic goal expressions

Non-basic goal expressions are proven in OCML using the following rules:

`(and A B)`	This expression is satisfied if both A and B can be proven in the current model.
`(or A B)`	This expression is satisfied if either A or B can be proven in the current model.

`(=> A B)`	This expression is satisfied if either A cannot be proven, or, if B can be proven in each environment which is a solution to A .
`(<=> A B)`	This expression is satisfied if both (=> A B) and (=> B A) can be proven.
`(not A)`	This expression is satisfied if A cannot be proven in the current model.
`(exists schema-or-var A)`	This expression is satisfied if A can be proven in the current model.
`(forall schema-or-var (=> A B))`	This expression is satisfied if either A cannot be proven, or, if B can be proven in each environment which is a solution to A.

Thus, the proof mechanism supported by OCML is not complete with respect to first-order logic statements. In particular, disjunctions can only be proved by proving each clause separately and negated expressions are only proved by default.

9 Bibliography

[Motta99] Enrico Motta. *Reusable Components for Knowledge Modelling*. IOS Press. 1-58603-003-5.

Function Index

Concept Index