An Overview of the OCML Modelling Language

Enrico Motta

Knowledge Media Institute
The Open University
Walton Hall, Milton Keynes, UK
E.Motta@open.ac.uk

Abstract. This paper provides an overview of the OCML modelling language: it illustrates the underlying philosophy, describes the main modelling constructs provided, and compares it to other modelling languages.

1. INTRODUCTION

OCML[1] was originally developed in the context of the VITAL project (Shadbolt et al., 1993) to provide operational modelling capabilities for the VITAL workbench (Domingue et al., 1993). Over the years the language has undergone a number of changes and improvements and in what follows we will provide an overview of the current version of the language (v5.1), illustrate its underlying philosophy and compare it to other knowledge modelling languages.

2. LANGUAGE TENETS

A number of ideas/principles have shaped the development of the OCML language. These are discussed in the following sections.

2.1. Knowledge-level modelling support.

The main goal of OCML is to support *knowledge-level modelling* (Newell, 1982; Fensel and Van Harmelen, 1994). In practice this role implies that OCML focuses on logical, rather than implementation-level primitives. Thus it provides mechanisms for expressing items such as relations, functions, rules, classes and instances, rather than arrays or hash tables. This approach is consistent with several other proposals for knowledge modelling (Gruber, 1993; Fensel and Van Harmelen, 1994). In the case of an operational language, such as OCML, this approach causes severe limitations in the support that the language can offer for efficient execution of application models. This problem can be (partially) addressed by providing a good compiler and by adding extra logical mechanisms for efficient reasoning - e.g. procedural attachments (Weyhrauch, 1980). Thus, while it is possible to specify and prototype knowledge models in OCML, the language does not aim to support efficient delivery of applications.

2.2. Support for the TMDA modelling framework

While OCML can be used to support various modelling approaches, such as CommonKADS (Schreiber et al., 1994b) or Components of Expertise (Steels, 1990), its design has been specifically informed by the *Task/Method/Domain/Application* (*TMDA*) framework (Motta, 1997; Motta and Zdrahal, 1997), which is illustrated in the next section.

2.2.1. Overview of the TMDA framework

The TMDA framework distinguishes between four generic types of components in an application model: *task knowledge*, *problem solving knowledge*, *domain knowledge*, and *application-specific knowledge.*

[1] The acronym "OCML" stands for Operational Conceptual Modelling Language.

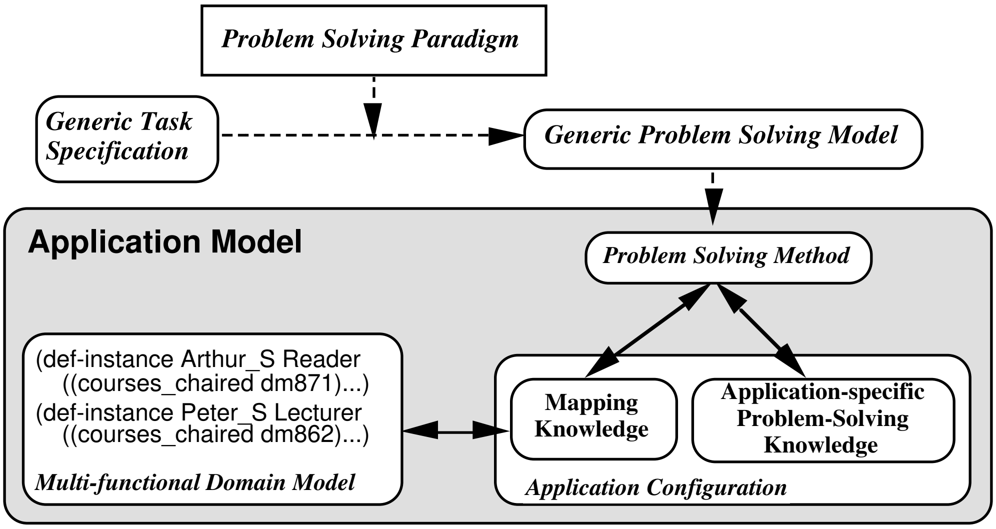

Figure 1. TMDA modelling framework

Task knowledge specifies a *generic task*, such as parametric design. This knowledge is expressed by means of a *task ontology* (Motta, 1997; Fensel et al., 1997).

Problem solving methods are domain-independent (and possibly task-independent) specifications of reasoning components. In our approach, we characterize problem solving methods as refinements of a *generic problem solving model*. This is associated with a task ontology, say T, and is constructed by instantiating a generic search model of problem solving in terms of T (Motta and Zdrahal, 1997).

Domain knowledge is knowledge associated with a particular application domain. Different approaches take different lines on the nature of domain knowledge and on its role in knowledge modelling, sharing and reuse. In particular, work on *multi-functional* knowledge bases (Murray and Porter, 1988) attempts to build large bodies of knowledge which, while domain-specific, are not committed to a particular task or problem solving method. By far the most famous of these large multi-functional knowledge bases is Cyc (Lenat and Guha, 1990), which comprises (or aims to comprise) the millions of notions which make up 'consensus reality'. The rationale for building multi-functional knowledge bases is quite simple: they provide a useful focus for reuse. As Murray and Porter (1988) point out, “The grid of potential knowledge bases has three dimensions: domain, task, and problem solving method. Building knowledge based systems for individual cells of this grid is both costly and short-sighted”.

The TMDA framework follows a reuse-oriented approach and therefore characterizes the domain component as specifying multi-functional, reusable domain models.

2.2.1.1. Application configuration knowledge

The knowledge interaction hypothesis (Bylander and Chandrasekaran, 1988) states that both the type of knowledge required by an application and its representation are strongly determined by the chosen task and/or method. Re-formulated in terms of the TMDA framework, this hypothesis indicates that both the knowledge acquisition process and the domain modelling schema are determined by the *method ontology* associated with the current problem solving method. In a sense this is trivially true. If we assume that a knowledge acquisition process starts with a pre-existing problem solving method and no domain component, then the quickest route to complete an application model is to acquire the knowledge required by the method and represent it in terms of the modelling schema specified by the method ontology. In a parametric design problem, this process will involve acquiring - among other things - the various parameters and constraints and representing them in terms of the relevant constructs provided by a particular parametric design method ontology. This approach was used by the researchers building knowledge acquisition tools according to the role-limiting approach (Marcus, 1988) and by us when tackling the VT elevator design problem in the VITAL project (Motta et al., 1996). A more interesting scenario is one in which not only a problem solving method, but also a pre-existing domain knowledge base (say a database of employees) already exists for the application domain in question (say office allocation). In this case we would like to reuse not just the problem solving method but also the existing domain model. The knowledge interaction hypothesis poses two objections here: the pre-existing domain model will not comprise the 'right' knowledge, and its modelling schema will be inappropriate. The second objection is easy to address. Any representation mismatch can be addressed by adding appropriate *mapping mechanisms* (Gennari et al., 1994), which allow the problem solver to retrieve and make use of the relevant definitions from the domain model. Of course, this can be in principle inefficient. However, this is not an issue here, as we are not looking for efficient representation solutions but for modelling constructs enabling us to build knowledge-level models of applications out of reusable components.

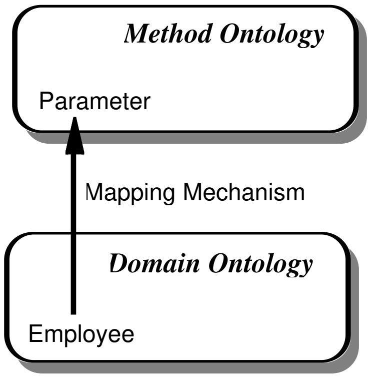

Figure 2. Mapping domain to method ontologies.

The other objection concerns the possible mismatch between the knowledge required by the method and that embedded in the multi-functional domain knowledge base. For example, in an office allocation problem it is important to acquire knowledge about office preferences and allocation requirements. This knowledge is very much application-specific and therefore cannot be part of a multi-functional domain knowledge base. Hence, it needs to be acquired on an application-specific basis. In order to account for these two modelling situations, formulating the relevant mapping mechanisms and acquiring application-specific knowledge, the TMDA framework comprises a fourth type of component, *application configuration knowledge*. In contrast with the other three components this does not specify a class of reusable components. Rather, it provides the ‘glue’ for integrating reusable problem solving and domain elements.

2.2.1.2. Usable vs. reusable components

The approach presented here tries to reconcile the trade-off between *usability* and *reusability*. The more generic a model, the more reusable it is. The more specific a model, the more usable it is, although in a restricted space of applications. The proposed framework addresses these issues at the knowledge level, i.e. in terms of modelling solutions, rather than efficiency considerations. At the knowledge level, the problem can be reduced to one of different degrees of coupling between different components of a knowledge model. In particular, the TMDA framework applies the following design principles.

- **Strong coupling between generic tasks and problem solving methods.** Problem solving methods are designed to perform efficient problem solving and they are typically designed with a class of tasks in mind. A close coupling between a generic method and a generic task is therefore not so much a requirement, as a consequence of the *raison d'être* of problem solving methods. In the TMDA framework this ‘natural’ coupling is strengthened by using the choice of a problem solving paradigm as a mechanism for providing a principled approach to developing a generic problem solving model and method ontology for a given problem type.

- **Weak coupling between generic problem solving models and multi-functional domain knowledge**. By definition, multi-functional knowledge is domain knowledge which characterizes task-independent aspects of a domain, i.e. domain knowledge which can be used in many different ways. For the sake of reusability this knowledge is modelled in a task and method-independent way. It follows that only weak coupling of multi-functional and problem solving knowledge can be supported. This weak coupling is expressed by means of mapping mechanisms.

- **The knowledge interaction problem can be tackled at the knowledge level by introducing an application-configuration component**. In particular, the assumption here is that the interaction problem can be operationalized at the knowledge level in terms of the activities of acquiring application-specific knowledge and establishing the appropriate mappings between the problem solving and domain components.

Thus, the application configuration component comprises two types of application-specific knowledge, *mapping knowledge*, which is required to integrate weakly coupled components, and *application-specific problem solving knowledge*, which is required by the chosen problem solving method and needs to be acquired for each application. This second type of knowledge has often a heuristic connotation. However, not all application-specific problem solving knowledge is heuristic. For instance, the cost function in design applications is typically application-specific but not heuristic. Therefore, it seems to us that the often-encountered dichotomy between "the conceptual aspects of a domain theory and the *heuristic annotations*" (Steels, 1990) is of limited use. The main issue when constructing reusable models is to separate the domain elements which can be reused across applications - i.e. multi-functional domain knowledge - from those which are application-specific. The latter might include, but cannot be reduced to, heuristic knowledge.

2.2.2. Logical and ontological support for the TMDA framework

In order to support the TMDA framework, OCML should provide primitives supporting the specification of domain models, task and method components, and mapping knowledge. Earlier versions of the language (Motta, 1995) provided special-purpose modelling constructs for expressing tasks and methods. The current version does away with these task and method-specific constructs and only provides a basic set of domain modelling facilities; the extension to the language required to specify tasks and methods is then defined as a particular representation ontology, the *task-method* ontology (Motta, 1997). The advantage of this approach is that it separates the core set of logical primitives (the OCML kernel) from additional, framework-specific epistemological commitments. Thus, only special-purpose primitives for defining mapping knowledge are required, in addition to 'standard' domain modelling capabilities.

2.3. Compatibility with emerging standards

The development of the OCML language has been driven by several pragmatic considerations. An important one concerns the compatibility with established or emerging standards. In particular, OCML includes domain modelling facilities which closely mirror a significant subset of those provided by Ontolingua (Gruber, 1993). This approach allows easy translation between OCML and Ontolingua. Moreover, these capabilities allow Ontolingua users to use OCML as a kind of 'operational Ontolingua' providing theorem proving and function evaluation mechanisms for Ontolingua constructs. Such facilities are interactive and therefore support incremental model construction, rather than the 'batch mode' style of interaction associated with the translation approach used for operationalizing Ontolingua models in LOOM (Mac Gregor, 1991) or in other languages.

OCML can also be used for KADS-style modelling: the relevant modelling constructs required for representing KADS interpretation models (Wielinga et al, 1992) can be defined by configuring the OCML task-method ontology for the KADS framework.

2.4. Integration of formal/informal/operational modelling

Different users require different modelling styles. Moreover, the same user may require different kinds of modelling support at different stages of the knowledge engineering life-cycle. For instance, it is quite common to develop an informal application model first and formalize and operationalize this at a later stage - see for instance (Motta et al., 1994; 1996). Therefore, it is important to provide a language (or an integrated set of languages) which can support various modelling styles, such as formal, informal and operational.

Operational modelling is supported in OCML by providing interpreters for evaluating control and functional terms as well as theorem proving facilities. OCML can also be used for formal specifications: formal semantics can be given to functional terms and logical expressions by translating them to the equivalent Ontolingua expressions[2].

Finally, informal modelling is supported by means of a graphical notation and the provision of pseudo-OCML code - see (Motta, 1997) for examples.

2.5. Support for quick prototyping of knowledge models

An important principle underlying the design of OCML concerns the provision of facilities supporting rapid prototyping of knowledge models. This stance is grounded on both fundamental and pragmatic reasons. The former are related to the experimental nature of artificial intelligence (AI) and, in particular, knowledge-based systems (KBSs). Although much knowledge engineering work over the past decade - e.g. the KADS and VITAL projects - can be seen as a reaction to the rapid prototyping approach dominating AI in the previous decade, the complexity of AI problems and the exploratory nature of much AI programming[3] makes AI systems much less amenable than conventional programs to a structured development approach, based on a rigid separation between a formal, non-executable specification and an implementation. Non-executable specifications are of limited utility when dealing with problems which have non-polynomial complexity. AI problems are typically ill-structured, which means that problem solvers necessarily need to search for a solution. Search is reduced by acquiring knowledge about the search space. This knowledge can be acquired either by doing (i.e. by navigating the search space) or by being told (i.e. by eliciting the relevant problem solving knowledge). In both situations quick prototyping is crucial to support knowledge acquisition in a situated problem solving context[4].

Incidentally, while executable specifications are especially important for KBSs, similar arguments have been raised also in the software engineering field. Fuchs (1992) points out that early validation is crucial to improve software reliability and refutes the argument made by Hayes and Jones (1989) that executable specifications over-constrain the space of possible software designs. In particular Fuchs argues that executable, logic-based, declarative languages can adequately express functionalities at a level of abstraction similar to that of non-executable specification languages. It is interesting to note that Fuchs emphasizes that the key to maintain a high level of abstraction within an executable specification language is that this should be declarative and able to search. In AI in general, and KBSs in particular, this property is crucial, not just to formalize a problem but also to model the problem solving behaviour of any feasible implementation.

---

[2] No formal semantics is provided for the control language. Moreover, OCML also includes some extra-logical facilities, such as procedural attachments.

[3] Such 'exploratory nature' of AI programming is the main reason for the curious phenomenon that happens when an AI problem is given a solution. At that point it is common to hear sceptics pointing out that the problem in question was not an AI problem in the first place, or that the particular solution is not an instance of AI. In our view there is a simple, generic explanation for this phenomenon. AI is about navigating large search spaces and understanding their structure. Once a search space has been explored and we know how to reach solutions, then it is natural to stop perceiving the problem as AI. The debate surrounding Deep Blue provides a typical instance of the phenomenon. For years one of the main goals of AI was to produce a chess program capable of beating the world champion. Now that the goal has been reached (to a limited extent), then the goal itself ceases to be AI. In a sense this is true. Once Deep Blue is able to manage the complexity of the search space involved in chess games, then the problem loses its connotation of "decision-making under uncertain conditions", which is the defining feature of AI.

[4] For instance it is interesting to note that several contributions to the Sisyphus-II initiative (Schreiber and Birmingham, 1996) report 'discoveries' related to the nature of the domain knowledge and the behaviour of the problem solver, which emerged only after the implementation of the end system. And this (fairly typical) phenomenon emerged in the context of an application, the VT elevator design problem (Yost and Rothenfluh, 1996), which has been extensively analysed and reconstructed several times!

Therefore, we believe that it is essential for a knowledge modelling language to be able to support rapid prototyping and incremental development and testing. To this purpose OCML provides operational modelling support and extends the language with non-logical facilities such as procedural attachments, which can be used for carrying out efficient proofs and evaluations, or for interfacing to pre-existing code. A simple example of the role of procedural attachments can be seen in the case of basic modelling primitives for list manipulation and numeric computation. While the relevant ontologies can provide formal specifications of numbers and lists, when prototyping a model it is convenient to extend a formal model of arithmetic with attachments which can effectively perform numerical calculations.

3. TYPES OF CONSTRUCTS IN OCML

OCML supports the specification of three types of constructs: *functional* and *control terms*, and *logical expressions*. 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*. 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. Finally, 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 usual operators - `and`, `or`, `not`, `=>`, `<=>` - and quantifiers - `forall` and `exists`. Operational semantics is provided for all operators and quantifiers.

4. BASIC DOMAIN MODELLING IN OCML

In the previous section we have introduced the three types of constructs which are supported by OCML. In this section we will go down at a more fine-grained level of description and illustrate the various primitives which are provided in OCML to support the specification of logical expressions, functional and control terms. In particular, OCML provides mechanisms for defining *relations*, *functions*, *classes*, *instances*, *rules* and *procedures*.

4.1. OCML 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.

4.1.1. Relation specification options

From a formal semantics point of view the purpose of a relation option is to help characterizing the extension of a relation. Table 1 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.

Table 1. Relation specification options in OCML

4.1.2. 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. Table 2 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

Table 2. Operationally-relevant relation options in OCML

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, see table 1, they are equivalent from a constraint checking point of view: 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 - see table 1. 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 function).

4.1.3. A meta-option for non-operational specifications

As shown above, OCML provides a number of relation options (specifically: `:iff-def`, `:sufficient`, `:def` and `:constraint`), 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))))))
```

4.2. OCML 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 (i.e. logical) 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.[5]

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,

5 These constraint-checking mechanisms can be switched off, if they are not required.

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.

4.3. OCML classes and instances

OCML also supports the specification of classes and instances and the inheritance of slots and values through *isa hierarchies*. OCML provides support for the usual slot specification machinery which is found in frame-based languages, e.g. type restrictions, slot values, default slot values, cardinality restrictions, etc. Instances are simply members of a class. An example of instance definition, taken from the Sisyphus-I domain model (Motta et al., 1994), 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 an OCML slot can have multiple values. In this case `harry_c` works both with `angy_w` and `werner_l`.

4.4. Object-oriented and relation-oriented approaches to modelling

When describing classes and instances we 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 we 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.

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 uses 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 an '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 both of these are relations, it is possible to characterize them by means of the relation options discussed in section 4.1. 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))))
```

4.5. The generic Tell-Ask interface

OCML provides a generic assertion-making primitive, `tell`, which provides a uniform mechanism for asserting *facts*[6], independently of whether these refer to slot-filling assertions, class instances, or simply relation instances. For example, we 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, we can find out about the projects in which `harry_c` is involved, by using the Lisp

---

6 Formally a *fact* (or *assertion*) is a *ground relation expression*. As discussed in section 3, a relation expression is an expression such as (<r> <arg$_1$>....<arg$_n$>), where <r> is a relation and arg$_i$ is a functional term. A ground expression (or term) is an expression (or term) which does not contain variables.

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 an OCML model 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 rules*[7] - and will then try these in sequence, to generate one or more solutions to the query - see (Motta, 1997) for more details on the OCML proof system.

4.6. OCML 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. 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))))
```

4.7. Rule-based reasoning in OCML

4.7.1. Backward rules

OCML also supports the specification of *backward* and *forward* rules. A backward rule consists of a number of *backward clauses*, each of which 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.

---

7 See section 4.7.1 for a description of OCML backward rules.

4.7.2. Forward rules

OCML also allows the user to define forward rules. 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 relation or function specification constructs.

5. 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. These services are described in detail in (Motta, 1997).

6. 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 task and method specification 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. In the next sections we will discuss the two kinds of mapping constructs supported by OCML: *relation mapping* and *instance mapping*.

6.1. Instance mapping

In figure 2 we showed 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 (YQT-member). 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 mix two different types of concepts. In particular, parameters and YQT members maintain different set 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` and associating this to I.

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 approach is used in the definition of relation `maps-to`, shown below. This definition makes use of the function `denotation`[8], which specifies a non-operational association between an object and its denotation.

```
(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))))
```

6.2. 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. Note the use of relation `maps-to` to retrieve the parameter associated with each particular YQT member.

---

8 The function `denotation` is part of the Ontolingua base ontology.

```
(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`. This 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. BASE ONTOLOGIES

OCML also provides the basic support for constructing models out of pre-existing components. 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 sub-ontologies. When conflicts are detected (e.g. the same concept is defined in two different sub-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. For instance 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 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 is used 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. 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.

8. COMPARISON WITH OTHER LANGUAGES

In the previous section we compared the base ontologies provided by OCML and Ontolingua and emphasized that the differences between them have mainly to do with the conceptual requirements imposed by the TMDA framework on OCML and with its operational nature. If we compare OCML and Ontolingua purely as modelling languages, it is easy to see that their main difference has to do with the fact that while Ontolingua is concerned exclusively with ontologies - i.e. term specification - OCML aims to model behaviour as well. For this reason OCML also provides support for defining control terms. Apart from this aspect, the current version of OCML closely mirrors many of the constructs in Ontolingua. Thus, while OCML extends Ontolingua in various respects, it is possible (if reductive) to view OCML as an environment for prototyping Ontolingua models, thus moving away from the batch-oriented, translation-based operationalization model suggested for Ontolingua (Gruber, 1993).

The design philosophy underlying the KARL language (Fensel, 1995) has many points in common with OCML. In particular, both KARL and OCML are operational modelling languages and are therefore suitable for rapid prototyping of knowledge level models. Moreover, both KARL and OCML are part of comprehensive knowledge engineering frameworks providing methodological support for KBS development. Needless to say, there are also differences. KARL is an executable, formal specification language where the emphasis is on formalization: its main strength is the provision of a formal semantics for its modelling constructs. The design philosophy of OCML has more to do with pragmatic considerations: its main goal is to provide a flexible modelling environment able to support different approaches to modelling: rapid prototyping and structured development, executable and non-executable constructs, formal and informal specifications. The two languages also differ in terms of the modelling frameworks they employ: KARL is based on the KADS four-layer framework, while OCML (more precisely the task-method ontology) is based on the TMDA framework. An important difference is also that while the primitives in the KARL language closely reflect the KADS approach, the OCML kernel is approach-neutral. Its commitment to the TMDA framework is defined by means of the appropriate ontology. This approach has the advantage of flexibility: different modelling frameworks can then be supported through the specification of the relevant ontologies.

Recently a new version of KARL (*New-KARL*) has been proposed (Angele et al., 1996), which does away with the strong KADS-centred organization used by KARL and focuses instead on the specification of task-method structures and ontology mappings. Thus, New-KARL subscribes to a modelling framework which has much more in common with the OCML task ontology than the one underlying KARL. However, in contrast with OCML, New-KARL also 'hardwires' these task and method-centred primitives in the language itself, rather than in a particular ontology.

Other formal specification languages exist for KADS models - see (Fensel and van Harmelen, 1994) for an overview. While the formal details of these languages of course vary, similar conclusions to those drawn above can be reached when comparing them to OCML: these languages tend to emphasize formal aspects and are based on a KADS approach. OCML emphasizes operationality and flexibility and does not presuppose (although it can support) a KADS approach. Indeed, the original raison d'être for OCML was to provide an operational alternative to a formal specification language, K_{BS}SF (Jonker and Spee, 1992), in the context of the VITAL workbench (Domingue et al., 1993).

Among the informal notations available for knowledge modelling the most notable is the CML language (Schreiber et al., 1994a), which supports ontological specifications and the construction of Common KADS models. CML supports the definition of various constructs, including concepts, attributes, tasks, methods, relations, structures and expressions. Obviously, the main difference between CML and OCML is that the former is only meant to be an informal notation while the latter is a fully operational language. Another important difference is that CML is committed to supporting the Common KADS framework, while the kernel of the OCML language is framework-independent. In addition to the KADS-related commitments, CML also embodies other modelling commitments: it provides primitives for representing structures and part-of relations. This approach has both advantages and disadvantages. On the plus side, it extends the range of modelling primitives provided by the language and supports notions which occur frequently in conceptual modelling. On the other hand there are two possible problems with this approach. The first one has to do with embedding ontological commitments in a conceptual modelling language. In particular different approaches to modelling structure and aggregation can be found in the literature - e.g. compare the analysis by Martin and Odell (1995) with that by Lenat and Guha (1990). Embedding one particular approach in the kernel of a conceptual modelling language prevents users from extending the language according to an alternative approach. The second problem is caused by the provision of different levels of description - i.e. *logical*, *epistemological* and *conceptual* (Brachman, 1979) - within the same formalism. Much work in knowledge representation over the past twenty years has focused on identifying the different levels at which knowledge representation languages can be specified (Brachman, 1979; Guarino 1994). In particular the paper by Brachman clearly illustrates that much of the confusion surrounding the field of knowledge representation in the seventies was caused by the fact that researchers were comparing formalisms which were situated at different levels. Thus, it seems to us that including ontological primitives (e.g. structures) in a language characterized at the logical level is a potential source of confusion. This is especially the case with informal languages, given that eventual ambiguities are not explained by the underlying formal theory.

The LOOM language (MacGregor, 1991) is strictly speaking a knowledge representation rather than knowledge modelling language (i.e. it also includes symbol-level representation constructs). However, its formal kernel - i.e. the *terminological* and *assertional* components - provides purely logical and epistemological primitives and therefore it can be used for knowledge modelling and ontology specification. The main feature of LOOM is its powerful classification mechanism, which integrates a sophisticated concept definition language with rule-based reasoning. This approach allows a wider range of inferences to be drawn than those available from 'traditional' frames+rules systems such as KEE. The existence of a powerful classifier is an important advantage that LOOM maintains over OCML. On the other hand, viewed purely as a knowledge modelling language, LOOM exhibits a number of limitations.[9] When building knowledge models it may be necessary to make statements about the model being developed, which do not have a direct inferential purpose. For this reason OCML allows the inclusion of non-operational statements and supports the specification of axioms about the current model. In contrast, LOOM only provides operational constructs which might restrict the usability of this language for knowledge modelling. Another possible problem related to LOOM is that while it supports different knowledge representation paradigms (frames, rules, message-passing) it integrates the various constructs according to a classification-centred viewpoint. While this approach obtains nice results in terms of inferential capabilities, it nevertheless can be a constraint for the knowledge analyst, who is forced to frame the current problem within a classification-centred framework. In contrast with this approach, OCML provides a number of alternative modelling constructs, e.g. rules, functions, classes and procedures which, while integrated, are themselves 'primitive modelling components' and can be used within different modelling approaches.

9. CONCLUSIONS

The OCML language is meant to provide a useful tool for knowledge modelling. Its primary purpose is to provide operational knowledge modelling facilities and, to this end, it includes interpreters for functional and control terms, as well as a proof system which integrates inheritance with backward chaining, function evaluation and procedural attachments. While the emphasis here is on operationality, of course we recognize that different styles of knowledge modelling ought to be supported. Therefore OCML supports full first-order logic definitions and allows the user to explicitly distinguish non-operational from operational definitions.

---

[9] The following points should not be construed as criticisms of the LOOM language. This is primarily a knowledge representation language and should be of course judged with respect to knowledge representation criteria. However, given the high-level of support provided by the language, its 'organic relationship' with the Ontolingua effort, and its sound theoretical basis, it makes sense to consider it as a plausible candidate for knowledge modelling. Indeed LOOM has recently been used for ontological work (Swartout et al., 1996).

Moreover, it supports an extensive set of Ontolingua constructs and therefore can be used as an interpreter for Ontolingua definitions.

OCML has been used extensively to support the construction of a library of reusable components for parametric design (Motta, 1997; Motta and Zdrahal, 1997), as well as in a number of other application areas, such as medicine, geology and material science. The system is implemented in Common Lisp and is available on various platforms. Finally, a web-based tool is being developed (Domingue, 1998), which supports collaborative editing of OCML models over the World-Wide-Web.

10. REFERENCES

Angele, J., Decker, S., Perkuhn, R. and Studer, R. (1996). Modelling Problem Solving Methods in New KARL. In B. Gaines and M. Musen (Editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*. Banff, Alberta, Canada.

Brachman, R. J. (1979). On the Epistemological Status of Semantic Networks. In N. V. Findler (Editor), *Associative Networks: Representation and Use of Knowledge by Computers*. Academic Press, New York, pp. 3-50.

Brachman, R. J., Fikes, R. E. and Levesque, H. J. (1985). KRYPTON: A Functional Approach to Knowledge Representation. In R. J. Brachman and H. J. Levesque (Editors). *Readings in Knowledge Representation*. Morgan Kaufmann, Los Altos, CA.

Bylander, T. and Chandrasekaran, B. (1988). Generic Tasks in Knowledge-Based Reasoning: The Right Level of Abstraction for Knowledge Acquisition. In B. Gaines and J. Boose (Editors), *Knowledge Acquisition for Knowledge-Based Systems - volume 1*, pp. 65-77. Academic Press, London.

Domingue, J. (1998). Tadzebao and WebOnto: Discussing, Browsing, and Editing Ontologies on the Web, John Domingue, UK. Paper submitted to the *Eleventh Workshop on Knowledge Acquisition, Modelling and Management*. Banff, Alberta, Canada. April 18th-23rd, 1998.

Domingue, J., Motta, E. and Watt, S. (1993). The Emerging VITAL Workbench. In N. Aussenac, G. Boy, M. Linster, J-G Ganascia and Y. Kodratoff (Editors). *Knowledge Acquisition for Knowledge-Based Systems - EKAW '93*. Lecture Notes in Artificial Intelligence, LNCS 723, Springer-Verlag. 1993.

Fensel, D. (1995). *The Knowledge Acquisition and Representation Language KARL.* Kluwer, Dordrecht.

Fensel, D. and Straatman, R. (1996). Problem solving methods: Making Assumptions for Efficiency Reasons. In N. Shadbolt, K. O'Hara, and Schreiber, G. (Editors). *Advances in Knowledge Acquisition - EKAW '96*. Lecture Notes in Artificial Intelligence, 1076. Springer-Verlag, Heidelberg.

Fensel, D. and van Harmelen, F. (1994). A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise. *The Knowledge Engineering Review, 9(2).*

Fensel, D., Motta, E., Decker, S., and Zdrahal, Z. (1997). The use of Ontologies for Specifying Tasks and Problem Solving Methods: A Case Study. In R. Benjamins and E. Plaza (Editors). *Knowledge Acquisition, Modelling, and Management. Proceedings of the 10th European Workshop, EKAW '97.* Lecture Notes in Artificial Intelligence 1319, Springer-Verlag.

Fikes, R. E. and Kehler, T. (1985). The Role of Frame-Based Representation in Reasoning. *Communications of the ACM, 28(9)*, September 1985.

Fuchs, N. E. (1992). Specifications are (preferably) executable. *Software Engineering Journal*. pp. 323-334. September 1992.

Genesereth, M. R. and Fikes, R. E. (1992). Knowledge Interchange Format, Version 3.0. *Technical Report Logic-92-1*, Computer Science Department, Stanford University.

Genesereth, M. R. and Nilsson, N. J. (1988). *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Los Altos, CA.

Gennari, J. H., Tu, S. W., Rothenfluh, T. E. and Musen, M. A. (1994). Mapping Domains to Methods in Support of Reuse. *Proceedings of the 8th Banff Knowledge Acquisition Workshop*, Banff, Canada.

Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition, 5(2).*

Guarino, N. (1994). The Ontological Level. In R. Casati, B. Smith and G. White (Editors), *Philosophy and the Cognitive Science*. Holder-Pichler-Tempsky, Vienna, Austria.

Hayes, I. J. and Jones, C. B. (1989). Specifications are not (necessarily) executable. *Software Engineering Journal, 4(6)*, pp. 330-338. November 1989.

Jonker, W. and Spee, J. W. (1992). Yet Another Formalization of KADS Conceptual Models. In Th. Wetter, K.-D. Althoff, J. Boose, B.R. Gaines, M. Linster and F. Schmalhofer (Editors) *Current Developments in Knowledge Acquisition - EKAW '92*, pp. 112-32. LNAI 599, Springer-Verlag, Berlin.

Lenat, D.B. and Guha, R.V. (1990). *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project*. Addison-Wesley, Reading, MA.

Levesque, H. J. (1984). Foundations of a functional approach to knowledge representation. *Artificial Intelligence 23(2)*, pp. 155-212.

MacGregor, R. (1991). Using a Description Classifier to Enhance Deductive Inference. *Proceedings of the 7th IEEE Conference on AI Applications*. Miami, Florida, February 1991.

Marcus S. (Editor) (1988). *Automatic Knowledge Acquisition for Expert Systems.* Kluwer Academic, Boston, MA.

Martin, J. and Odell, J. J. (1995). *Object-Oriented Methods: A Foundation.* Prentice-Hall, Englewood Cliffs, New Jersey.

Motta, E. (1995). KBS Modelling in OCML. *Proceedings of the fifth Workshop on Modelling Languages for KBS*, Vrije Universiteit Amsterdam, 30-31 January, 1995.

Motta, E. (1997) Reusable Components for Knowledge Models. PhD Thesis. Knowledge Media Institute. The Open University. UK

Motta, E. and Zdrahal, Z. (1996). Parametric Design Problem Solving. In Gaines, B. and Musen, M. (editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´96)*, Banff, Canada, November 9th-14th, 1996.

Motta, E. and Zdrahal, Z. (1997). An approach to the organization of a library of problem solving methods which integrates the search paradigm with task and method ontologies. Submitted for publication. Available from http://kmi.open.ac.uk/~enrico/papers/ijhcs_psm.ps.gz. July 1997.

Motta, E., O'Hara, K., and Shadbolt, N. (1994). Grounding GDMs: A Structured Case Study. *International Journal of Human-Computer Studies 40(2)*, pp. 315-347.

Motta, E., Stutt A., Zdrahal Z., O'Hara K. and Shadbolt N. (1996). Solving VT in VITAL: a Study in Model Construction and Knowledge Reuse. *International Journal of Human-Computer Studies 44(3/4)*, pp. 333-371.

Murray, K. and Porter, B. (1988). Developing a Tool for Knowledge Integration: Initial Results. *Proceedings of the 3rd Banff Knowledge Acquisition Workshop*, Banff, Canada.

Newell A. (1982). The knowledge level. *Artificial Intelligence, 18(1)*, pp. 87-127.

Schreiber, A.T. and Birmingham W.P. (Editors) (1996). Special Issue of the *International Journal of Human-Computer Studies, 44(3/4).* March/April 1996.

Schreiber, A.Th., Wielinga B. J., Akkermans, H., van de Velde, W., and Anjewierden, A. (1994a). CML: The CommonKADS Conceptual Modelling Language. In L. Steels, A. T. Schreiber, and W. van de Velde (Editors), *A Future for Knowledge Acquisition, Proceedings of the 8th European Knowledge Acquisition Workshop*. Springer Verlag, LNAI 867, pp. 283-300.

Schreiber, A. T., Wielinga B. J., de Hoog, R., Akkermans, H., van de Velde, W., and Anjewierden, A. (1994b). CommonKADS: A Comprehensive Methodology for KBS Development. *IEEE Expert*, December 1994, pp. 28-37.

Shadbolt, N., Motta, E., and Rouge, A. (1993). Constructing Knowledge Based Systems. *IEEE Software*, *10(6)*, pp. 34-38.

Steels, L. (1990). Components of Expertise. *AI Magazine, 11(2)*, pp. 29-49.

van Harmelen, F. and Balder, J. R., (1992). $(ML)^2$: A Formal Language for KADS Models of Expertise. *Knowledge Acquisition, 4(1)*, pp. 127-161.

Weyhrauch, R. W. (1980). Prolegomena to a theory of mechanized formal reasoning, *Artificial Intelligence (13)1-2*, pp. 133-170

Wielinga B. J., Schreiber A.T. and Breuker J., (1992). KADS: A Modelling Approach to Knowledge Engineering. *Knowledge Acquisition 4(1)*, pp. 5-53.

Yen, J., Neches, R. and MacGregor, R. (1988). Classification-based Programming: A Deep Integration of Frames and Rules. Technical Report ISI/RR-88-213. USC/Information Science Institute, March 1988.

Yost G.R. and Rothenfluh T.R. (1996). Configuring elevator systems. *International Journal of Human-Computer Studies, 44(3/4)*, pp. 521-568.