CML: The CommonKADS Conceptual Modelling Language

Guus Schreiber[1], Bob Wielinga[1], Hans Akkermans[34], Walter Van de Velde[2] and Anjo Anjewierden[1]

[1] University of Amsterdam, Social Science Informatics
Roetersstraat 15, NL-1018 WB Amsterdam, The Netherlands
[2] Free University of Brussels, AI Lab
Pleinlaan 2, B-1050 Brussels, Belgium
[3] Netherlands Energy Research Foundation ECN
P.O. Box 1, 1755 ZG Petten, The Netherlands
[4] University of Twente. Information Systems Department
P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. We present a structured language for the specification of knowledge models according to the CommonKADS methodology. This language is called CML (Conceptual Modelling Language) and provides both a structured textual notation and a diagrammatic notation for expertise models. The use of our CML is illustrated by a variety of examples taken from the VT elevator design system.

1 Introduction

In this paper we describe a highly structured, semi-formal notation for the specification of CommonKADS expertise models [16]. This notation is called CML, short for Conceptual Modelling Language. The CML described here covers domain knowledge (including the specification of ontologies), inference knowledge, task knowledge as well as problem solving methods. Below, we discuss the various constructs of the CML for each of these categories in subsequent sections. The practical use of the CML is illustrated by various examples taken from the VT elevator design domain [8, 17], as reanalyzed with the help of the CommonKADS methodology [11].

In an appendix we give the full set of BNF grammar rules defining the syntax of the *textual* CML constructs. In addition to textual CML definitions, we provide a *graphical* notation allowing the knowledge engineer to concisely present the main features of an expertise model in a set of diagrams. We note that the graphical notation is not intended to replace the textual description, as it often abstracts from details that are present in this textual description. On the other hand, a diagrammatic notation is very useful in representing, explaining and communicating knowledge structures in a way accessible to both knowledge engineers and users. Thus, textual and diagrammatic notations have complementary functions. Our graphical notation follows as much as possible the notations used in software engineering, especially with respect to the domain knowledge where we closely follow the object-oriented OMT (Object Modelling Technique) notation proposed by Rumbaugh et al. [9]. In the final discussion section we comment on the relationships and differences of the CommonKADS CML with the OMT and Ontolingua [5] specification languages.

2 Domain Knowledge

The domain knowledge in an expertise model consists of three parts:

1. *Ontology definitions*: sets of type definitions of domain constructs, such as concepts and relations.
2. *Ontology mappings*: a description of how types of one ontological theory can be mapped onto types in another ontology.
3. *Domain models*, denoting knowledge base partitions containing domain expressions that use a set of ontology definitions.

We use the term ontology to denote a "specification of a conceptualisation" [6]. In earlier publications [15, 16] we distinguished two types of ontologies: (i) the *model ontology*, and (ii) the *domain ontology*, defining respectively the PSM-specific and the domain-specific conceptualisations. Since then, it has become clear that it may be useful to make additional distinctions, e.g. within the model ontology. See for a discussion on types of ontologies and their role in knowledge engineering [14, 11]. For the purpose of defining the CML we assume that there is a need to describe various types of ontologies, without committing ourselves to what these ontologies precisely are.

2.1 Ontology definitions

An ontology is defined through the specification of a number of types or "constructs". The CML provides a number of representational primitives each of which is briefly discussed in this section: *concept*, *attribute*, *expression*, *structure* and *relation*. The example definitions are taken from the VT model ontology [11]. The existence of this ontology can be defined in CML as follows (see also the appendix):

```
ontology VT model ontology;
  description:
      This ontology contains domain-independent type definitions for describing
      structural properties of the VT knowledge base [17].;
  definitions:
      < see the sample concept, attribute, expression, structure
      and relation definitions>
end ontology
```

Concept The notion of *concept* is used to represent a class of real or mental objects in the domain being studied. The term "concept" corresponds roughly to the term "entity" in ER-modelling and "class" in object-oriented approaches.

Every concept has a *name*, a unique string which can serve as an identifier of the concept, possible super concepts (multiple inheritance is allowed), and a number of *properties*. a property is a (possibly multi-valued) function into a value set. A number of value-sets are assumed to be pre-defined, such as strings, integers, natural numbers, real numbers and booleans. A newly defined value-set can be a range of integers or reals or an enumeration of strings. For the definition of value sets, see the appendix. Relations of a concept with other concepts, attributes or expressions should be modeled separately with CML relation definitions (see further).

Below two example concepts definitions found in the VT domain are given.

```
concept component;
  description:
      components are part types of an artefact. Instances in the VT domain are:
      "elevator", "car buffer", "car guiderail", etc.;
end concept
```

```
concept component-model;
  description:
      represents a particular model of a component, e.g. "car buffer OH1", "car
      buffer OM14", etc. Component models often have fixed attribute values, such
      as weight, physical dimensions, etc.;
end concept
```

Fig. 1 shows the graphical notation for concepts and sub-type relations between concepts. Concepts are indicated with rectangles. Three notations are provided for the sub-type relation between concepts. In principle, the OMT notation with the triangle should be preferred. The other two are included because many knowledge engineers use them as a convenient shorthand. Fig. 5 (see further) shows the graphical representation of concepts in another ontology (the `VT domain-ontology`).

Attribute An attribute is a reification of a function. One can see it as a shorthand for a concept with no internal structure and with a single "value" property. Attributes are graphically represented as rectangles, just as concepts, but with the name of the value set written as a subscript (see Fig. 4 for an example).

```
attribute attribute-slot;
  description:
      An attribute-slot is used to represent component attributes, such as weight,
      width, length, etc.;
  value-set: number ∨ string;
end attribute
```

Expression The notion of expressions as a domain modelling construct is introduced because these occur often in "domain rules" or "domain axioms". An important aspect of the domain modelling enterprise is to describe the structure of these domain rules. This type of domain description is currently lacking in many KBS development projects. The *expression* construct provides a suitable way of modelling the structure of domain knowledge in which simple expressions such as $age(patient) > 65$ and $temperature(patient) = high$ appear.

The general form of expressions is $< operand >< operator >< value >$ where:

- *operand* is a either an attribute or some property of a concept,
- *operator* is one of $=, \neq, <, \leq, >, \geq, \in, \subset, \subseteq, \supset, \supseteq$,
- *value* is a sub-set of the value-set of the function (i.e. attribute. concept property).

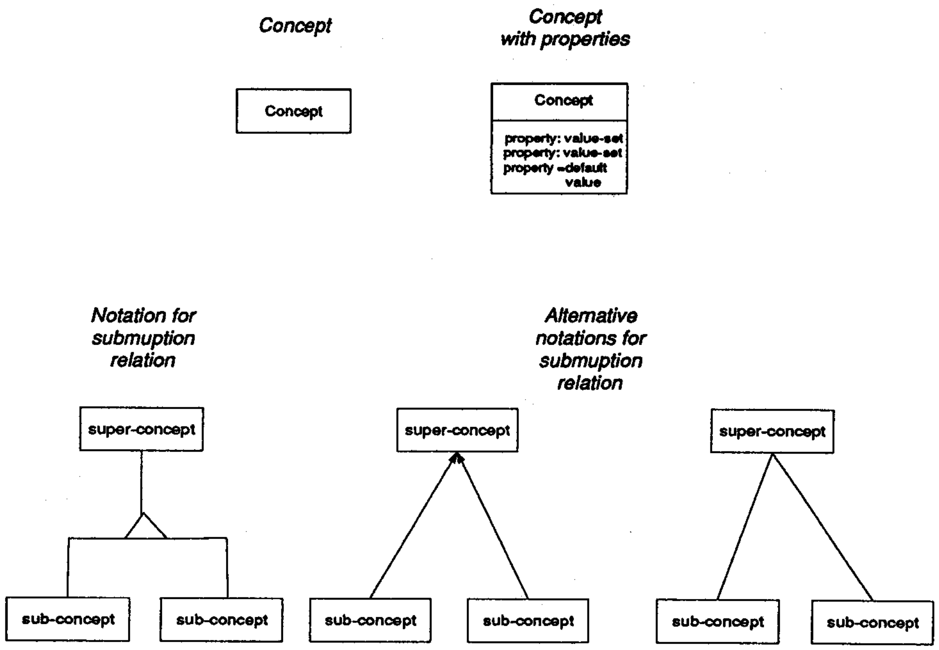

Fig. 1. Graphical notation for concepts (optionally with or without property definitions) and three alternative notations for sub-type relations between concepts. The left-most is the OMT notation and should be preferred

Expressions can be restricted to a particular property of a concept, and to a particular subset of operators. An example of an expression in the VT domain is an equality expression about an `attribute-slot`:

```
expression attribute-slot-expression;
  description:
      Represents a simple expression about an attribute slot: e.g. "height = 28.75";
  operand: attribute-slot
  operators: = ;
end expression
```

In the CML description of an expression the specification of operators may be omitted. In that case it is assumed that all legal operators on the values denoted by the properties can be used. For example, if an expression is defined on a property with a numeric value set, the set of possible operators is $=, \neq, <, \leq, >, \geq$, The set operators $(\in, \subset, \subseteq, \supset, \supseteq)$ are typically used with value sets that consist of a set of symbols. Expressions are represented graphically through an oval with the name of the expression and an arrow indicating the operand. Fig. 2 shows the notation for the two types of expression operand definitions.

Structure The notion of structure is used in the CML to describe objects with an internal structure that the knowledge engineer does not want to describe (at this

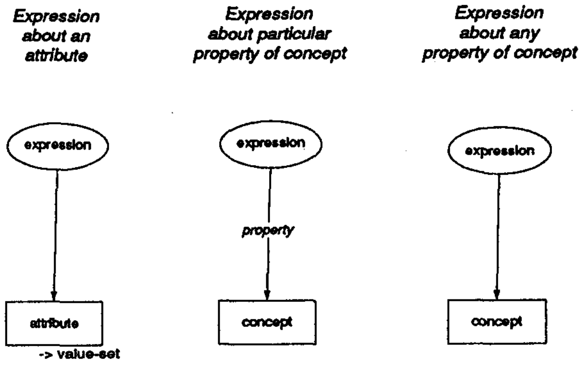
Expression about an attribute
Expression about particular property of concept
Expression about any property of concept
expression
expression
expression
property
attribute
-> value-set
concept
concept

Fig. 2. Expressions are represented graphically through an oval and an arrow indicating the operand. These operands can be of three types: an attribute, some particular property of a concept, or any property of a concept.

moment) in detail. An example in the VT domain for which one could decide to model it as a structure is a *constraint expression*. Constraint expressions represent mathematical and logical dependencies between system variables. The knowledge engineer might want to introduce this as an explicit type in the ontology, without being forced to write down a full syntactical description of the form of constraint expressions. The CML description of structures is similar to that of concepts, but with an additional `form` slot in which the knowledge engineer can give a textual description of this class of objects:

```
structure constraint-expression;
  form:
      any (mathematical, logical) dependency between attribute-slot values.;
end structure


structure calculation;
  subtype-of: constraint-expression;
  form:
      a constraint expression of the form:
      <attribute-slot> = <mathematical-formula>;
end structure
```

The two examples define the notion of (ii) a constraint expression, without detailing the precise structure of the underlying formula, and (ii) a calculation as a sub-type of constraint expression implying a particular type of formula, again without going into syntactical details. The graphical representation of a structure is a rectangle (see Fig. 4).

Relation The notion of *relation* is a central construct in modelling a domain. In the CML we allow various forms of relations to cater for the specific requirements imposed by knowledge-based systems. The relation construct is used to link any type of objects to each other, including concepts, attributes, expressions, structures and relations.

The grammar rules in the appendix specify the CML for defining relations. The CML supports two types of relation arguments: (i) a single object (e.g concept, attribute, expression, structure, another relation), and (ii) a set of such objects. An example of the second type of argument would be modelling causal relations as a binary relation with a "causes" argument that refers to a set of expressions about some state variable. Relations can themselves also have properties. The classical example of such a property is the wedding date of two married people. Below three example relation definitions of the VT domain are shown.

```
binary-relation has-constraint;
  description:
      binary relation linking a component to a constraint expression specifying some
      dependency between attribute-slot values of this components or its sub-parts.;
  inverse: constraint-on;
  argument-1: component;
  argument-2: constraint-expression;
end binary-relation


binary-relation has-attribute;
  description:
      binary relation linking components and component models to attribute slots.;
  argument-1: component ∨ component-model;
  argument-2: attribute-slot;
  axioms:
      ∀ c:component m:component-model a:attribute-slot
        has-attribute(c, a) ∧ has-model(c, m)
        → has-attribute(m, a);
end binary-relation


relation fixed-model-value;
  description:
      binary relation defining fixed values for attribute slots of a component-model;
  argument-1: component-model;
  argument-2: attribute-slot-expression;
  axioms:
      ∀ m:component-model a:attribute-slot
        fixed-model-value(m, a = v) → has-attribute(m, a);
end relation
```

The first relation, `has-constraint`, links a **component** to a constraint expression specifying some dependency between attribute-slot values of this components or its sub-parts. The second relation defines the link between components and their models on the one hand and attribute slots on the other hand. The (optional) `axioms` field states that attribute slots defined for components also apply to their models. The third relation shows the use of an expression construct in a relation definition. This relation (`fixed-model-value`) links a component model to an equality expression about an attribute slot. This relation can be used to model fixed values for attributes of a component model, e.g. `fixed-model-value(carbuffer OH1, height = 28.75)`.

Graphically relations are represented as diamonds, just as in ER modelling. For binary relations there is an alternative, directional, representation (see Fig. 3). Properties of relations are represented as arrows from relations to value sets. Set arguments of relations are indicated with the join symbol ⋈. Multiple types for an argument are represented through a split line.

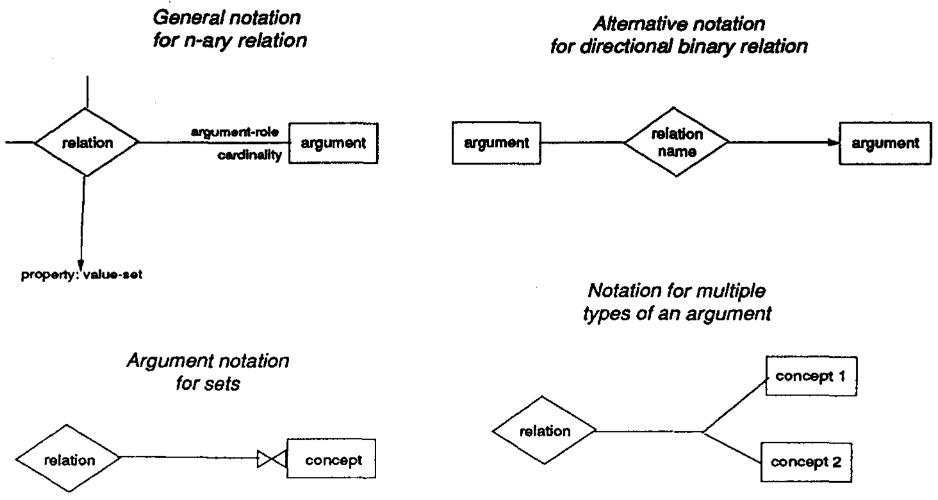
General notation for n-ary relation
relation
argument-role
cardinality
argument
property: value-set
Alternative notation for directional binary relation
argument
relation name
argument
Argument notation for sets
relation
concept
Notation for multiple types of an argument
relation
concept 1
concept 2

Fig. 3. Graphical notation for relations. For binary relations there is an alternative representation. Properties of relations are represented as arrows from relations to value sets. Arguments of relations can be either single constructs (concepts, attributes, expressions, relations) or sets of these constructs. Sets are indicated with the join symbol ⋈. Multiple types for an argument are shown through a split line

Fig. 4 shows the graphical representation of the CML VT definitions given in this section, plus an additional `has-model` relation. .

2.2 Ontology mappings

In the situation where a KBS is built from scratch it is possible to define one ontology (a model ontology in CommonKADS terms), and view the actual knowledge base as a pure *instantiation* of that ontology. In the light of efforts to share and/or reuse knowledge bases and ontologies, this approach turns out to be insufficient. For example, in the VT example there was an existing domain knowledge base with its own ontology. Some typical fragments of this knowledge base are shown in Fig. 5. We call this knowledge base the VT domain ontology. To use this knowledge base given the VT model ontology, of which parts were defined in this paper, one has to specify a mapping procedure that shows how constructs defined in the domain ontology should be mapped onto the model ontology.

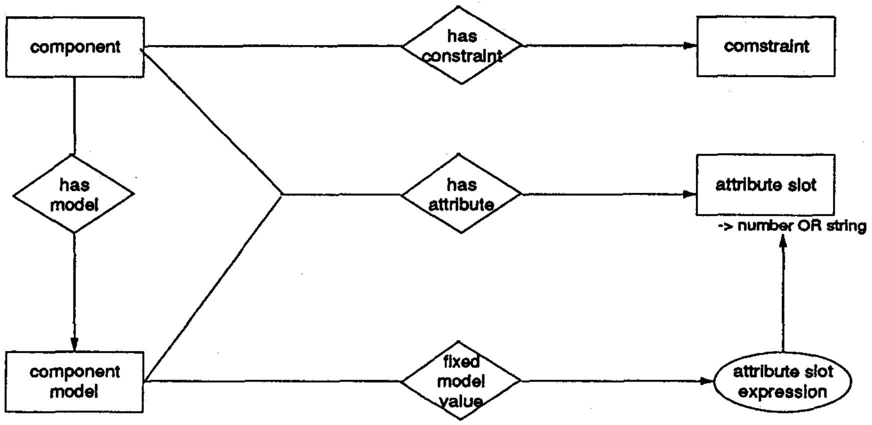

Fig. 4. Graphical representation of the sample VT definitions

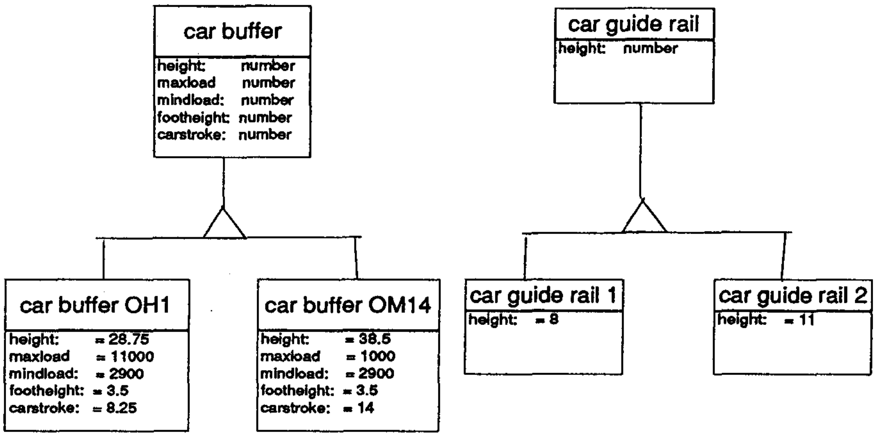

Fig. 5. Domain-specific concepts in VT domain ontology. The notation "property = value" is used to show default values

The purpose of the CML construct `ontology-mapping` is to allow for the (informal) definition of such mappings. An example of the mapping between the VT domain ontology in Fig. 5 and the part of the model ontology shown in Fig. 4 is given below. The mappings are defined here in natural language. In the context of the work on ML^2, [13] a rewrite formalism was developed to specify such mappings in a formal way. Work on the nature of ontology mappings is underway (see, for example, [4]). We expect the CML specification of ontology mappings to be refined in the near future.

```
ontology-mapping
  from: VT domain ontology;
  to: VT model ontology;
  mappings:
    Concept with no superconcept in the domain ontology
    ↦ component in the model ontology

    Concept with superconcept in the domain ontology
    ↦ component-model in the model ontology

    Sub-type relation between concepts
    ↦ tuple of has-model relation

    Property of concept
    ↦ an instance of attribute-slot plus a tuple of has-attribute

    Default value of concept
    ↦ tuple of fixed-model-value relation, if the concept is a
      component-model;
end ontology-mapping
```

More details on the mapping shown above is given elsewhere [11, Appendix A].

2.3 Domain models

A domain model is a coherent collection of expressions about a domain that represents a particular viewpoint defined in an ontology. The domain model may therefore embody certain assumptions that are specific for the ontology that it uses.

In the CML a domain model is defined as a composite object. It is defined through a number of parts which contain one or more sets of objects (instances, tuples). The graphical notation for domain models (see Fig. 6) is inspired by the way data stores are represented in data-flow diagrams, because these are intuitively quite similar. Domain models can be viewed as a sort of "knowledge stores". Domain models have an internal structure, represented through aggregate-part links.

3 Inference Knowledge

In this section we define the CML for the specification of inferences and provide a new graphical notation for showing the data dependencies between inferences (the inference structure).

3.1 Inference specification

Names of inferences represent the role these inferences play in solving the problem. Inference names are thus goal-oriented. In addition, we specify the *operation type*: the abstract operation that is performed on some ontology, similar to the inference ontology in KADS-I. For the moment we use the formalised set of inferences defined by Aben [1993] as the basis for describing operation types.

For each role, a mapping is specified to the domain knowledge. For static roles, we may also indicate which domain model should be accessed to find this body of

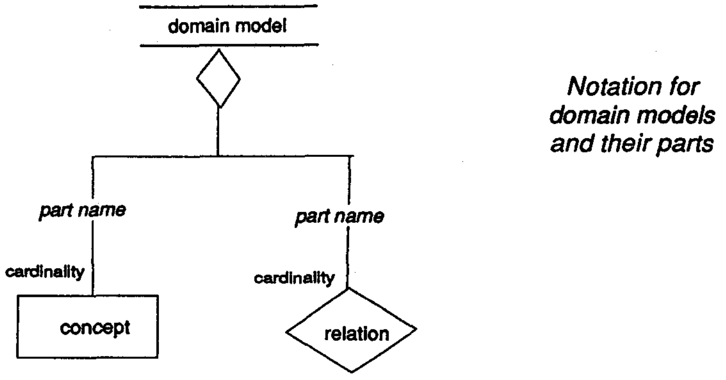

Fig. 6. A domain model is represented as a "knowledge store" with an internal part-of structure. The "diamond" symbol is overloaded and indicates in this context an aggregate-part relationship

knowledge. Dynamic roles are supposed to be part of the overall working memory of the problem solver and are thus not directly linked to a domain model.

We show two example inferences from the VT application. The *select-parameter* inference selects a parameter to which a value can be assigned. The static roles refer to constraint formulae in two domain models which for reasons of space were not defined in the previous section (see [11]).

```
inference select-parameter;
  operation-type: select;
  input-roles:
      parameter-set → set of attribute-slots;
      parameter-assignments → set of tuples <attribute-slot, value, dependencies>;
  output-roles:
      parameter → single attribute-slot;
  static-roles:
      formulae ∈ domain models initial-values and calculations;
  spec:
      Select a parameter from the skeletal model that has not been assigned a value
      and for which the preconditions that the domain knowledge (i.e. the formulae)
      poses for computing the value are fulfilled. A precondition is the fact that a
      value of some other parameter should be known.
      Formulae in the domain model initial values are evaluated as soon as possible.
      A heuristic ordering of components is used to rank the set of parameters. (see
      [17, Start of Sec. 5]);
end inference
```

The second example inference is *specify value*. This inference uses the constraints in two domain models to compute a value for a selected parameter.

```
inference specify-value;
  operation-type: compute;
  input-roles
```

```
        parameter → attribute-slot;
        parameter-assignments → set of tuples <attribute-slot, value, dependencies>;
    output-roles:
        parameter-assignment → single tuple <attribute-slot, value, dependencies>;
    static-roles:
        formulae ∈ domain models initial-values and calculations;
    spec:
        Specify the value of a parameter by interpreting the formulae and identify the
        parameters that were used in this specification (the dependencies);
end inference
```

3.2 Inference structure

Inference structures are among the most frequently-used ingredients of KADS. In almost any presentation of an application of KADS, the description of the inference structure plays a dominant role. In this section we define some additional graphical notations to remove a number of ambiguities in inference structures.

Transfer tasks Transfer tasks are treated in the expertise model as black-box functions. Inferences and transfer tasks together form the lowest level of functional decomposition in the expertise model. One could say that transfer tasks are basic functions that do not make any inferences in the domain knowledge. Thus, it seems appropriate to include transfer tasks in an inference structure. A rounded-box notation is used to distinguish transfer tasks from inferences.

Role element vs. set Another issue that has arisen with respect to inference structures concerns the nature of roles. A role constitutes a functional name for a set of domain objects that can play this role. Some inferences operate on or produce *one particular* object, others work on a *set* of these objects. This can lead to ambiguities in inference structures, for example if one inference produces one object and another inference works on a set of these objects, possibly generated by some repeated invocation of the first inference. The graphical CML notation allows for making this distinction explicit: a ⋈ symbol indicates that the input or output should be interpreted as a set of objects playing this role.

Role names Another problem arises from the names given to roles. Some role names constitute a general name for objects involved in carrying out a task. For example, *observable*, *finding*, and *hypothesis* are such general role names. In addition, more specialised role names are also used. Often, such names are a specialisation of the general categories, e.g. *test observable*, *discriminating observable.* If one looks upon a role as a container of objects applying that role, a specialised name represents a label for a subset of objects in a container.

Specialized names such as *test observable* are useful and make the inference structure easier to interpret. On the other hand, some inferences may operate on the general category (e.g. observable). One would like to be able to specify both general and specialized role names and still be able to show clearly the dependencies between inferences. We support this in the graphical notation by making the subset structure of containers explicit through a "subset/superset" link between roles. An example of this notation is shown in Fig. 7.

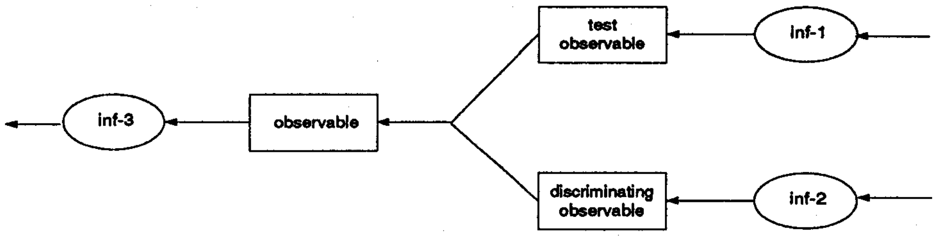

Fig. 7. Introducing general and specialized role names in an inference structure. An object taking the general role *observable* can be generated by two inferences, each using a specialized name for this role (*test observable* and *discriminating observable*). The link between the three roles indicates that *test observable* and *discriminating observable* are in fact subsets of the objects playing the role of *observable*

Static roles Inference structures only used to show the dynamic roles that are being manipulated by an inference. Sometimes, it is also useful to show what type of domain knowledge the inference uses to derive the output from the input (cf. [7]). This domain knowledge is specified through the static roles.

One could argue that this is an unwanted extension of the inference structure, as inferences are in fact domain-independent generalizations of the application of domain knowledge. However, it can be useful at some points during knowledge engineering to make the nature of the domain knowledge explicit, although this destroys the domain-independence of an inference structure. We use a double arrow to indicate the domain knowledge used by an inference, if felt necessary.

Inference and role annotations Optionally, the inferences can be annotated with the operation type of the inference, e.g. the inference "specify-value" can be annotated with the operation type "compute" (see Fig. 9).

Also, dynamic role names may be annotated with the name that is used for this role in the task knowledge. These task role names (see the next section) can be more informative for a user. An example of this type of annotation can be found in Fig. 9 where the role *parameter-assignments* has as a subscript *extended-model* which is the name for this role from the task (goal-directed) point of view.

Fig. 8 summarises the graphical notation for inference structures. Fig. 9 shows the inference structure for the two VT inferences specified previously.

4 Task Knowledge

4.1 Task specification

The knowledge category *task knowledge* describes how a goal can be achieved through a task. A task specification consists of two parts: the *task definition* and the *task body*.

The task definition describes *what* needs to be achieved. It is a declarative specification of the goal of the task. The *task definition* consists of:

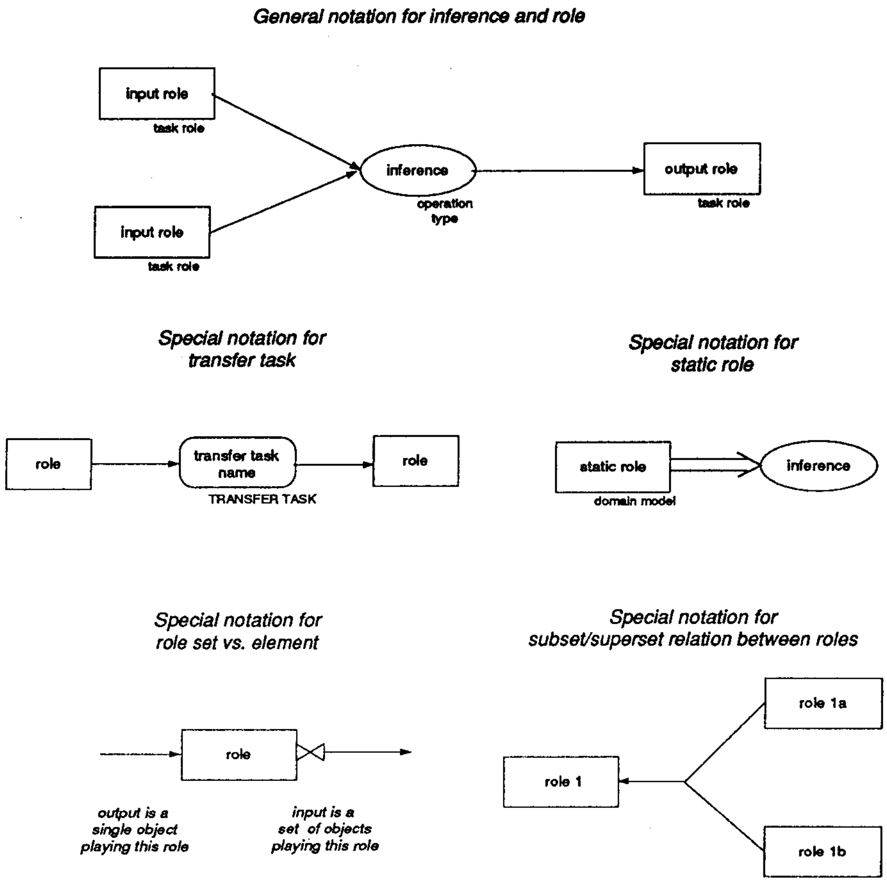
General notation for inference and role
input role
task role
input role
task role
inference
operation type
output role
task role
Special notation for transfer task
role
transfer task name
TRANSFER TASK
role
Special notation for static role
static role
domain model
inference
Special notation for role set vs. element
role
output is a single object playing this role
input is a set of objects playing this role
Special notation for subset/superset relation between roles
role 1
role 1a
role 1b

Fig. 8. Summary of the conventional used in the graphical representation of inference structures

Goal A textual description of the goal that can be achieved through application of the task.

Input/Output A definition of the roles that the task manipulates. This definition consists of a name and a textual description. The role is not directly bound to a domain type, as we do not want to have a direct coupling between task and domain knowledge. Instead, the task body (see below) specifies how a role is bound to other task roles and ultimately to dynamic roles of inferences. Only for inferences roles the mapping to domain knowledge is defined.

Task specification A description of the logical dependencies between the roles of the task (e.g. what is true after execution of a task, invariants). This description is optional.

The task definition of the VT design task describes the overall goal of the task and its I/O:

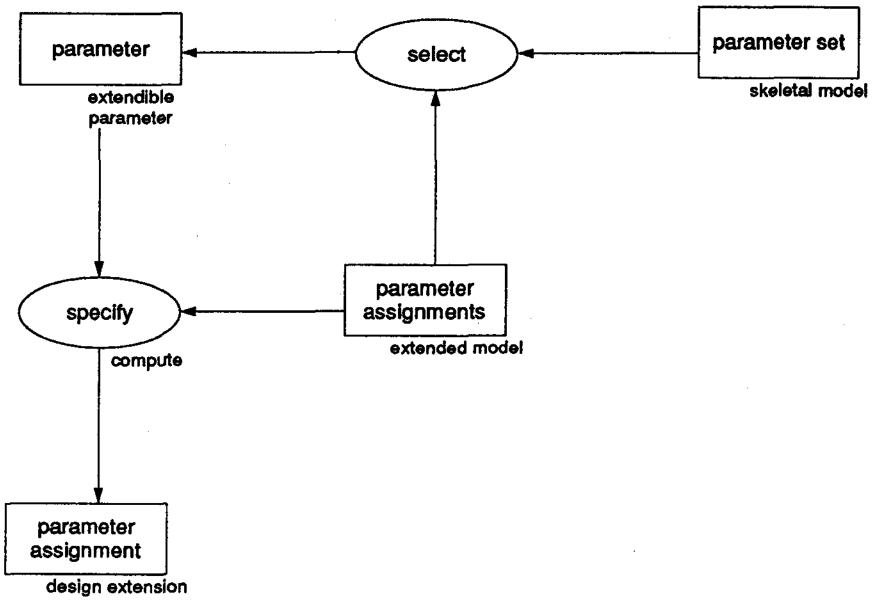

Fig. 9. Inference structure of the *propose* task. The names in the boxes denote inference roles. The annotated names represent the corresponding task role

```
task parametric-design;
  task-definition
    goal: find a design that satisfies a set of constraints;
    input:
      skeletal design: the set of system parameters;
      user-specs: set of parameter/value pairs;
    output:
      design: set of assigned parameters;
end task
```

The task body describes *how* this goal can be achieved. It is a procedural program, prescribing the activities to accomplish the task. We distinguish three different types of tasks, based on the nature of their task body:

Composite tasks are further decomposed in sub-tasks, e.g. diagnosis is decomposed into generate and test.

Primitive tasks are directly related to inferences. E.g. a primitive abstraction task could be the computation of all solutions of an *abstract* inference, given a particular data-set and a body of domain knowledge

Transfer tasks interact with the world, i.e., the user. The task body of a transfer task is not specified in the expertise model. It is contained in the communication model.

The CML description of the *task body* has the following subparts:

Task type One of *composite* or *primitive*. Task body descriptions of transfer tasks are not part of the expertise model.

Decomposition The sub-tasks that the task decomposes into. These can be other tasks or inferences.

Problem-solving method The PSM that was applied to achieve this decomposition.

Additional roles Additional data stores that are introduced by the decomposition.

Task control Structure The description of control over the sub-tasks to achieve the task.

Assumptions Additional assumptions implied by a decomposition, e.g. concerning certain knowledge structures, concerning the solution, etc. Assumptions are usually introduced by the underlying problem-solving method.

The sample task body of the VT design task is shown below:

```
task parametric-design;
  task-body
    type: composite;
    sub-tasks: init, propose, verify, revise;
    additional-roles:
      extended-design: current set of assigned parameters
        represented as a set of tuples < parameter, value, dependencies>
        where the dependencies constitute a set of parameters that were
        used in specifying the value of this parameter;
      design-extension: proposed new element of the extended model;
      violation: violated constraint;
    control-structure:
      configure(skeletal-design + user-specs → design) =
        init(user-specs → extended-design)
        REPEAT
          propose(skeletal-design + extended-design → design-extension)
          extended-design := design-extension ∪ extended-design
          verify(design-extension + extended-design → violation)
          IF some violation
          THEN revise(extended-design + violation → extended-design)
        UNTIL a value has been assigned to all parameters in the skeletal-design
        design := { <p, v> | <p. v, deps> ∈ extended-design };
end task
```

The control structure in the example above is written in procedural pseudo code. In principle however, the knowledge engineer is free to use any formalism that s/he finds best suitable for expressing control among sub-tasks. In real-time applications, for example, one could opt for a state transition formalism.

4.2 Task structure

In the process of engineering an expertise model, it is often useful to visualise the current set of tasks as a provisional "inference structure". This is a provisional structure in the sense that the "inferences" in such a diagram can in fact turn out to be (complex) tasks. An example of the use of such provisional inference structures can be found elsewhere [10]. We allow the knowledge engineer to use the same graphical conventions to represent these "task structures"[5] as for inference structures (see Fig. 8), with the exception that the "inferences" are not depicted as ovals but as boxes with rounded corners (similar to the convention used for transfer tasks). An example of a provisional inference structure for the VT example is shown in Fig. 10.

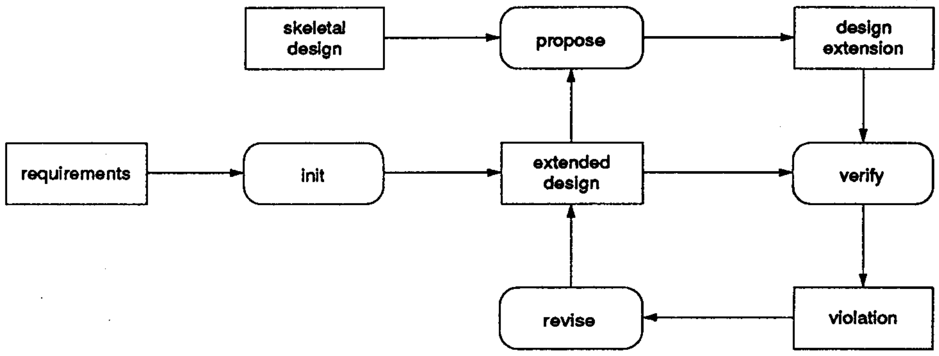

Fig. 10. Top-level data flow in the VT design task

In addition, it is often useful to show graphically the decomposition structure of tasks. An example of such a hierarchical decomposition can be found in Fig. 11. The problem solving methods that generated the decomposition can optionally be written on the lines connecting a task with its subtasks.

5 Problem Solving Methods

For many applications it suffices to specify the domain, inference and task knowledge to build a system. In that case the resulting KBS will have a fixed control structure, i.e. its behaviour is fixed. However, in some circumstances a more flexible form of control can be needed. In CommonKADS such flexibility can be achieved through the introduction of *problem solving knowledge* in the expertise model. Problem solving knowledge comes in two flavours: strategic knowledge and knowledge about problem solving methods [3]. The methods describe how a task definition can be given a task body that describes how to achieve the task goal. The strategic knowledge describes how methods are selected and applied in order to dynamically construct the task model. Below we give a CML definition of the top-level method for Propose-and-Revise.

```
PSM propose-and-revise
  input:
```

[5] The term "task structure" stands here just for a ' 'inference structure" representation of a set of tasks and should not be confused with the meaning of the word in KADS-I

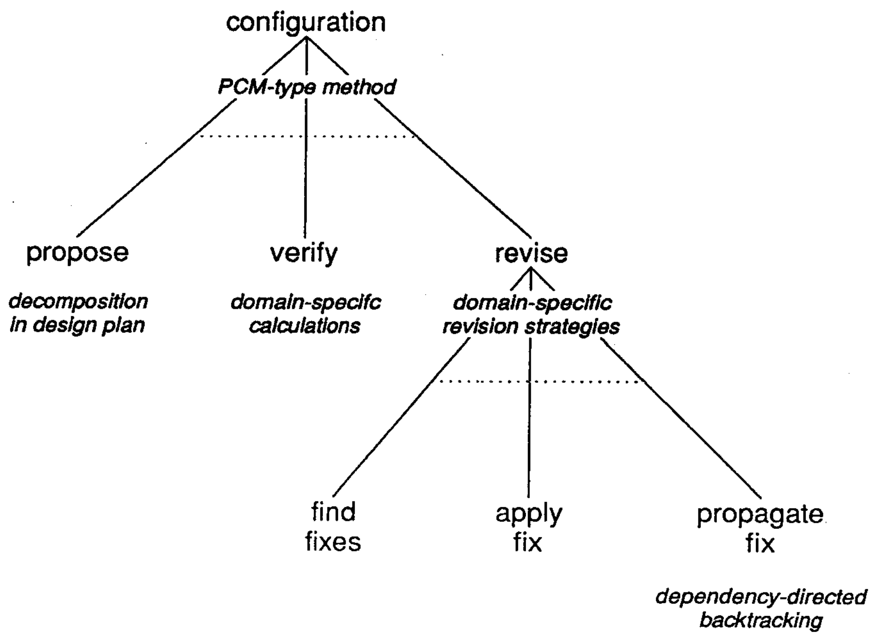
configuration
PCM-type method
propose
decomposition in design plan
verify
domain-specifc calculations
revise
domain-specific revision strategies
find fixes
apply fix
propagate fix
dependency-directed backtracking

Fig. 11. Task decomposition generated by P&R. The italic annotations characterise the methods that on which the various decompositions are based

```
    skeletal-design: set of parameters;
    requirements: set of operationalised user requirements;
output:
    extended-design: set of parameter assignments;
competence:
    ∀ req ∈ requirements:
      meets(extended-design, req) ∧
      consistent(extended-design, constraints);
sub-tasks:
    propose, verify, revise;
additional-roles
    par: element of the skeletal-design;
    design-extension: newly proposed parameter assignment ;
    violation: violated constraint ;
control-structure-template:
    init(requirements → extended-design)
    FOREACH par ∈ skeletal-design DO
      propose(par + extended-design → design-extension)
      extended-design := design-extension + extended-design
      verify(design-extension + extended-design → violation)
      IF some violation
      THEN revise(extended-design + violation → extended-design) ;
acceptance criteria:
        1. The requirements are operationalised in terms of initial values for some
        parameters
        2. Design knowledge can be represented as a set of constraints;
    end PSM
```

This problem solving method can be selected when a task definition has to be satisfied that has a goal that matches the *competence* of the method and when the acceptance criteria are met. When applied, the method will decompose the task into three subtasks (propose, verify and revise), introduce several intermediate roles that serve as place holders for intermediate results, and provides a template for a control regime over the sub tasks. This information is essentially sufficient to create a task body. The specification of the problem solving method given above, is largely informal. The CML does not provide explicit mechanisms to apply a method to a task definition, the method specification should be viewed as a structured way to write down knowledge about problem solving. A more formal account of problem solving methods is given in [2].

6 Discussion

In this paper we have described a language for specifying knowledge models: CML. The main advance of the CML is the facility to explicate the *ontology* of the domain knowledge. The ontology can be viewed as a *meta model* describing the structure of the domain knowledge. The mapping mechanism in CML allows the construction of a layered ontology in which higher layers represent abstract knowledge types. This facility is important since ontologies have structure: certain parts are based on generally accepted theory, other parts are based on common practice, useful interpretations or on task oriented notions. As a matter of principle we advocate to distinguish different partial ontologies that are based on different types of ontological commitments. Shareability and reusability of knowledge depend critically on the distinctions between the views that underly the different ontologies.

In many respects, the possibility to specify an explicit and structured ontology is similar to that of Ontolingua [5]. However, Ontolingua does not provide explicit mappings. On the other hand, Ontolingua is a fully formal language, while the CML is semi-formal. The semi-formal nature of CML is an advantage in early stages of the knowledge acquisition process: concepts and relations can be described in natural language. In later stages of KBS development a formal representation is needed. CML allows a formal representation, but does not prescribe a particular representation formalism. In [11] we have shown how CML can be used to specify and transform a formalised knowledge base. Also, tools exists to support structure-preserving operationalisation of CML descriptions in a dedicated executable environment [12].

CML has similarities with OMT [9] and other object-oriented specification frameworks, but offers -apart from the ontology- additional constructs such as *expressions*. Most object-oriented approaches do not separate the meta data model and the actual objects. In CML such a separation is possible through the use of the mapping mechanism. The inference and task layers of the CML have similarities with the control and functional views in conventional software engineering [18]. The graphical notations of CML are similar, but not identical, to the classical counterparts such as data flow diagrams. In CML the links between the different views are made explicit, while most classical approaches leave this link unspecified. In conclusion we claim that CML offers a number of new constructs and mechanisms that could also be of use in the field of classical software engineering.

Acknowledgements The CML notation is the result of a series of discussions in which many participants in KADS-II participated. Helpful comments were received from Christian Bauer, Frank van Harmelen, Andre van Naarden, Olle Olsson, Klas Orsvarn, Guy Saward, Peter Terpstra and André Valente, Manfred Aben built a parser for this CML and corrected in this way many small errors in the CML grammar.

The research reported here was carried out in the course of the KADS-II and KACTUS projects. These projects are partially funded by the ESPRIT Programme of the Commission of the European Communities as project number 5248 and 8145. The partners in the KADS-II project are Cap Gemini Innovation (F), Cap Programator (S), Netherlands Energy Research Foundation ECN (NL), Eritel SA (ESP), IBM France (F), Lloyd's Register (UK), Swedish Institute of Computer Science SICS (S), Siemens AG (D), Touche Ross MC (UK), University of Amsterdam (NL) and Free University of Brussels (B). The partners in the KACTUS project are Cap Gemini Innovation (F), LABEIN (ESP), Lloyd's Register (UK), STATOIL (N), Cap Programmator (S), University of Amsterdam (NL), University of Karlsruhe (D), IBERDROLA (ESP), DELOS (I), FINCANTIERI (I) and SINTEF (N). This paper reflects the opinions of the authors and not necessarily those of the the consortia.

References

1. M. Aben. CommonKADS inferences. ESPRIT Project P5248 KADS-II/M2/TR/UvA/041/1.0, University of Amsterdam, June 1993.
2. J. M. Akkermans, B. J. Wielinga, and A. Th. Schreiber. Steps in constructing problem-solving methods. In B. R. Gaines and M. A. Musen, editors, *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. Volume 2: Shareable and Reusable Problem-Solving Methods*, pages 29-1 – 29-21, Alberta, Canada, January 30 – February 4 1994. SRDG Publications, University of Calgary.
3. V. R. Benjamins. *Problem Solving Methods for Diagnosis*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, June 1993.
4. J.H Gennari, S.W Tu, T.E Rotenfluh, and M.A. Musen. Mapping domains to methods in support of reuse. In *Proceedings of the Knowledge Acquisition Workshop KAW-94*, Banff, Canada, 1994. SRDG Publications, University of Calgary.
5. T. Gruber. Ontolingua: A mechanism to support portable ontologies. version 3.0. Technical report, Knowledge Systems Laboratory, Stanford University, California, 1992.
6. T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5:199–220, 1993.
7. Marc Linster. *Knowledge acquisition based on explicit methods of problem–solving.* PhD thesis, University of Kaiserslautern, 1992.
8. S. Marcus and J. McDermott. SALT: A knowledge acquisition language for propose-and-revise systems. *Artificial Intelligence*, 39(1):1–38, 1989.
9. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
10. A. Th. Schreiber. Applying KADS to the office assignment domain. *International Journal of Man-Machine Studies*, 40(2), 1994. Special issue on Sisyphus 91/92 "Models of Problem Solving". In press.

11. A. Th. Schreiber, P. Terpstra, P. Magni, and M. van Velzen. Analysing and implementing VT using COMMON-KADS. In A. Th. Schreiber and W. P. Birmingham, editors, *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. Volume 3: Sisyphus II - VT Elevator Design Problem*, pages 44-1 – 44-29, Alberta, Canada, January 30 – February 4 1994. SRDG Publications, University of Calgary.
12. Peter Terpstra. An environment for application design. Deliverable DM7.5a, ESPRIT Project P5248 KADS-II/M7/DD/UvA/072/1.0, University of Amsterdam, 1994.
13. F. van Harmelen and J. R. Balder. $(ML)^2$: a formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1), 1992. Special issue: 'The KADS approach to knowledge engineering', reprinted in *KADS: A Principled Approach to Knowledge-Based System Development*, 1993, Schreiber, A.Th. *et al.* (eds.).
14. B. J. Wielinga and A. Th. Schreiber. Reusable and shareable knowledge bases: A european perspective. In *Proceedings International Conference on Building and Sharing of Very Large-Scaled Knowledge Bases '93*, pages 103–115, Tokyo, Japan, December 1-4 1993. Japan Informtation Processing Development Center.
15. B. J. Wielinga, W. Van de Velde, A. Th. Schreiber, and J. M. Akkermans. The KADS knowledge modelling approach. In R. Mizoguchi, H. Motoda, J. Boose, B. Gaines, and R. Quinlan, editors, *Proceedings of the 2nd Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 23–42. Hitachi, Advanced Research Laboratory, Hatoyama, Saitama, Japan, 1992.
16. B. J. Wielinga, W. Van de Velde, A. Th. Schreiber, and J. M. Akkermans. Towards a unification of knowledge modelling approaches. In Jean-Marc David, Jean-Paul Krivine, and Reid Simmons, editors, *Second Generation Expert Systems*, pages 299–335. Springer-Verlag, Berlin Heidelberg, Germany, 1993.
17. G. Yost. Configuring elevator systems. Technical report, Digital Equipment Corporation, 111 Locke Drive (LMO2/K11), Marlboro MA 02172, 1992.
18. E. Yourdon. *Modern Structured Analysis*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

A BNF specification of CML

The constructs of the CML are defined using BNF grammar rules. The conventions used in these grammar rules are summarised in Table A.

```
expertise-model        ::=     EXPERTISE-MODEL Application-name;
                                   domain-knowledge
                                   inference-knowledge
                                   task-knowledge
                                   psm-knowledge
                               END EXPERTISE-MODEL [Application-name;] .

domain-knowledge       ::=     DOMAIN-KNOWLEDGE
                                   < ontology-def |
                                     ontology-mapping-def |
                                     domain-model >*
                               END DOMAIN-KNOWLEDGE .

ontology-def           ::=     ONTOLOGY Ontological-theory-name;
                                   terminology
```

Construct	Interpretation
::= ⋆ + [ ] ⟨ ⟩ | .	Symbols that are part of the BNF formalism
X ::= Y.	The syntax of X is defined by Y
[X]	Zero or one occurrence of X
X⋆	Zero or more occurrences of X
X+	One or more occurrences of X
X | Y	One of X or Y (exclusive-or)
⟨ X ⟩	Grouping construct for specifying scope of operators e.g. ⟨ X | Y ⟩ or ⟨ X ⟩*.
SYMBOL	Uppercase: predefined terminal symbol of the language
Symbol	Capitalised: user-defined terminal symbol of the language
symbol	Lowercase: non-terminal symbols

Table 1. Synopsis of the notation used in BNF grammar rules

```
                                     [IMPORT: Ontological-theory-name
                                         <, Ontological-Theory-Name>*;]
                                     DEFINITIONS: domain-construct-def*
                                 END ONTOLOGY [Ontological-theory-name;] .

terminology               ::=    [ < DESCRIPTION: text; > ]
                                 [ < SYNONYMS: text; > ]
                                 [ < SOURCES: text; > ]
                                 [ < TRANSLATION: text; > ] .

domain-construct-def      ::=    object-def | relation-def | value-set-def .
object-def                ::=    atomic-object-def | constructed-object-def .
atomic-object-def         ::=    concept-def | attribute-def structure-def .
constructed-object-def    ::=    expression-def .

concept-def               ::=    CONCEPT Concept-name;
                                     terminology
                                     [SUB-TYPE-OF: Concept-name <, Concept-name >*;]
                                     [properties]
                                     [axioms]
                                 END CONCEPT [Concept-name;] .

properties                ::=    PROPERTIES: property-def <; property-def>* .
property-def              ::=    Property-name: value-set;
                                     [cardinality-def]
                                     [differentiation-def]
                                     [default-value-def] .
cardinality-def           ::=    CARDINALITY: [MIN natural] [MAX <natural | INFINITE>]; .
differentiation-def       ::=    DIFFERENTIATION-OF Property-name(Concept-name) .
default-value-def         ::=    DEFAULT-VALUE: Value; .
axioms                    ::=    AXIOMS: text; .

attribute-def             ::=    ATTRIBUTE Attribute-name;
```

```
                                terminology
                                [SUB-TYPE-OF: Attribute-name <, Attribute-name >*;]
                                [properties]
                                VALUE-SET: value-set;
                                [axioms]
                            END ATTRIBUTE [Attribute-name;] .

expression-def          ::=     EXPRESSION Expression-name;
                                terminology
                                [SUB-TYPE-OF: Expression-name <, Expression-name >*;]
                                [properties]
                                OPERAND: expression-operand
                                         <, expression-operand>*;
                                [OPERATORS: Operator-symbol <, Operator-symbol >*;]
                                [axioms]
                            END EXPRESSION [Expression-name;] .
expression-operand      ::=     Attribute-name |
                            SOME-PROPERTY-OF Concept-name |
                            Property-name OF Concept-name .

structure-def           ::=     STRUCTURE Structure-name;
                                terminology
                                [SUB-TYPE-OF: Structure-name <, Structure-name >*;]
                                FORM: text;
                                [properties]
                                [axioms]
                            END STRUCTURE [Structure-name;] .

relation-def            ::=     general-relation-def | binary-relation-def .

general-relation-def    ::=     RELATION Relation-name;
                                terminology
                                [SUB-TYPE-OF: Relation-name < Relation-Name >*;]
                                [properties]
                                ARGUMENTS: argument-def+
                                [axioms]
                            END RELATION [Relation-name;] .

binary-relation-def     ::=     BINARY-RELATION Relation-name;
                                terminology
                                [SUB-TYPE-OF: Relation-name;]
                                [properties]
                                [INVERSE: Relation-name;]
                                ARGUMENT-1: argument-def
                                ARGUMENT-2: argument-def
                                [axioms]
                            END BINARY-RELATION [Relation-name;] .

argument-def            ::=     argument-type <OR argument-type>* ;
                                [ARGUMENT-ROLE: Role-name;]
                                [cardinality-def] .
```

```
argument-type           ::=   domain-construct-type |
                              SET(domain-construct-type) |
                              LIST(domain-construct-type) .
domain-construct-type   ::=   built-in-type | user-defined-type .
built-in-type           ::=   OBJECT | CONCEPT | ATTRIBUTE |
                              EXPRESSION | RELATION .
user-defined-type       ::=   Concept-name | Attribute-name | Structure-name |
                              Expression-name | Relation-name .

primitive-type          ::=   NUMBER | INTEGER | NATURAL |
                              STRING | BOOLEAN | UNIVERSAL .

primitive-range         ::=   NUMBER-RANGE(Number, Number) |
                              INTEGER-RANGE(Integer, Integer) .

value-set               ::=   primitive-type | primitive-range |
                              Value-set-name | { String-value <, String-value>* } .

value-set-def           ::=   VALUE-SET Value-set-name;
                                  [TYPE: <NOMINAL | ORDINAL>;]
                                  [properties]
                                  < VALUE-LIST: String-value <, String-value>*; > |
                                  < VALUE-SPEC: < primitive-type | text > > ;
                              END VALUE-SET [Value-set-name;] .

ontology-mapping-def    ::=   ONTOLOGY-MAPPING
                                  FROM: Ontological-theory-name;
                                  TO:   Ontological-theory-name;
                                  MAPPINGS: text;
                              END ONTOLOGY-MAPPING .

domain-model            ::=   DOMAIN-MODEL Domain-model-name;
                                  USES: Ontological-Theory-Name;
                                  PARTS: part-def+
                                  [properties]
                                  [EXPRESSIONS: text;]
                                  [ANNOTATIONS: text;]
                              END DOMAIN-MODEL [Domain-model-name;] .

part-def                ::=   Part-name: part-element-def+ .
part-element-def        ::=   part-type ; [cardinality-def] .
part-type               ::=   SET(domain-construct-type) |
                              LIST(domain-construct-type) .

inference-knowledge     ::=   INFERENCE-KNOWLEDGE inference-def*
                              END INFERENCE-KNOWLEDGE .

inference-def           ::=   INFERENCE Inference-name;
                                  operation-type
                                  input-roles
                                  output-role
```

```
                                   static-roles
                                   inf-specification
                               END INFERENCE [Inference-name;] .
operation-type         ::=     OPERATION-TYPE: text; .
input-roles            ::=     INPUT-ROLES: dynamic-role-mapping+ .
output-role            ::=     OUTPUT-ROLE: dynamic-role-mapping+ .
static-roles           ::=     STATIC-ROLES: static-role-mapping* .
dynamic-role-mapping   ::=     Inference-role-name -> domain-references; .
static-role-mapping    ::=     domain-references IN Domain-model-name; .
domain-references      ::=     <domain-ref <, domain-ref >*> | text .
domain-ref             ::=     domain-construct-type |
                               SET(domain-construct-type) |
                               LIST(domain-construct-type) .
inf-specification      ::=     SPEC: text; .

task-knowledge         ::=     TASK-KNOWLEDGE task-description*
                               END TASK-KNOWLEDGE .

task-description       ::=     TASK Task-name;
                                   task-definition
                                   task-body
                               END TASK [Task-name;] .

task-definition        ::=     TASK-DEFINITION
                                   task-goal
                                   io-def
                                   [task-specification] .
task-goal              ::=     GOAL: text; .
io-def                 ::=     INPUT: role-description+
                               OUTPUT: role-description+ .
role-description       ::=     Task-role-name: text; .
task-specification     ::=     SPEC: text; .
task-body              ::=     TASK-BODY
                                   task-type
                                   decomposition
                                   [psm-ref]
                                   [additional-roles]
                                   [data-flow]
                                   control-structure
                                   [assumptions] .
task-type              ::=     TYPE: < COMPOSITE | PRIMITIVE > ; .
decomposition          ::=     SUB-TASKS: function-name <, function-name >*; .
psm-ref                ::=     PSM: Psm-name; .
function-name          ::=     Task-name | Inference-name .
additional-roles       ::=     ADDITIONAL-ROLES: role-description* .
control-structure      ::=     CONTROL-STRUCTURE: text; .
assumptions            ::=     ASSUMPTIONS: text; .

psm-knowledge          ::=     PROBLEM-SOLVING-METHODS psm-description*
                               END PROBLEM-SOLVING-METHODS .
```

```
psm-description        ::=    PSM Psm-name;
                                  io-def
                                  competence-spec
                                  decomposition
                                  [additional-roles]
                                  [data-flow]
                                  control-structure
                                  acceptance-criteria
                              END PSM [Psm-name] .
competence-spec        ::=    COMPETENCE: text; .
acceptance-criteria    ::=    ACCEPTANCE-CRITERIA: text; .
```