A Library of Components for Classification Problem Solving

Enrico Motta[1], Wenjin Lu[1]

Abstract

In this document we present a library of generic, reusable components whose purpose is to support the specification and operationalization of classification problem solvers. In addition, it is envisaged that this library will provide a test case for future version of the IBROW component brokering and configuration system. The library has been specified in the OCML modelling language, according to the IBROW framework for reuse. It comprises ontologies, task specifications and problem solving methods. It has been tested on a sample apple classification domain, independently developed by the University of Amsterdam. In the document we discuss our approach to modelling classification problem solving, we describe the various parts of the library, we illustrate examples from our application domain and we discuss related work and open issues.

Identifier:	**D1**
Class:	**Deliverable**
Version:	**1.0**
Date:	**7 September, 2000**
Status:	**Draft**
Distribution:	**Project Members**
Responsible partner:	**The Open University**

IBROW3 Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-1999-19005. The partners in this project are: the University of Amsterdam, co-ordinator (NL); the University of Karlsruhe (DE); The Open University (UK); the Spanish Council for Scientific Research-IIIA (ES); Stanford University (USA); Intelligent Software Components, S.A. (ES); and the Free University of Amsterdam (NL).

University of Amsterdam
Department of Social Science Informatics (SWI)
Roetersstraat 15
NL-1018 WB Amsterdam, The Netherlands
Tel: +31 20 525 6784; Fax: +31 20 525 6896
Contact person: V.R. Benjamins
E-mail: richard@swi.psy.uva.nl

University of Karlsruhe
Institute AIFB
D-76128 Karlsruhe
Germany
Tel: +49 721 608 3923; Fax: +49 721 608 6580
Contact person: R. Studer
E-mail: studer@aifb.uni-karlsruhe.de

The Open University
Knowledge Media Institute
Walton Hall
MK7 6AA, Milton Keynes, United Kingdom
Tel: +44 1908 653506; Fax: +44 1908 653169
Contact person: E. Motta
E-mail: e.motta@open.ac.uk

Spanish Council of Scientific Research (CSIC)
Artificial Intelligence Research Institute (IIIA)
Campus U.A.B.
08193 Bellaterra, Catalonia (Spain)
Tel: +34 935 809 570; Fax: +34 935 809 661
Contact person: Enric Plaza
E-mail: enric@iiia.csic.es

Stanford University
Stanford Medical Informatics
251 Campus Dr., Suite 215
94305-5479, Stanford, CA, USA
Tel: +1 650 723 6979; Fax: +1 650 725 7944
Contact person: M. Musen
E-mail: musen@smi.stanford.edu

Intelligent Software Components, S.A.
iSOCO Sant, Ctro.Neg. Can Castanyer
Crta. de Rubí, 88 bj. A
08190 St.Cugat (Barcelona), Spain
Tel +34 93 544 2048; Fax +34 93 589 5669
Contact person: E. Brieva
E-mail: erik@isoco.com

[7]**Free University Amsterdam**
Faculty of Sciences
Division of Mathematics and Computer Science
De Boelelaan 1081a
1081 HV Amsterdam, the Netherlands
Tel: +31 (0)6 51850619; Fax: +31 20 872 2722
Contact person: D. Fensel
E-mail: dieter@cs.vu.nl

Industrial advisory committee:

Daimler Chrysler (D)
Deutsche Telekom (D)
Bolesian (NL)
Unilever (NL)
IBM (JP)
Artificial Intelligence Applications Institute (UK)

1. INTRODUCTION

A knowledge-based system is typically described as a computer system which relies on a body of domain knowledge to make decisions under uncertain conditions and carry out complex tasks (Newell, 1982; Stefik, 1995). Research on *problem solving methods* (Benjamins and Fensel, 1998) and *ontologies* (Gruber, 1995) aims (among other things) to identify and formalise classes of generic components and reasoning patterns, which can be reused across different domains to support the robust development of knowledge-based systems. Loosely speaking, an ontology defines a conceptual vocabulary that can be used to describe a certain class of domains. For instance, the parametric design ontology developed by one of us (Motta, 1999) specifies the various concepts and relations required to model parametric design problems - e.g., parameters, constraints, requirements, design model, etc. A problem solving method describes a generic reasoning behaviour which can be reused in different applications. For instance, figure 1 shows the flow of inferences in the *heuristic classification model* defined by Clancey (1985), which can be used to solve classification and diagnostic tasks. According to this model, these tasks can be solved by a sequence of *abstraction*, *matching* and *refinement* operations, by which the available data are abstracted (for instance from a reading of the temperature of a patient we can infer that such temperature is high or low), matched to possible explanations (say, the patient has fever) and these can then be refined (e.g., the final diagnosis could be 'glandular fever').

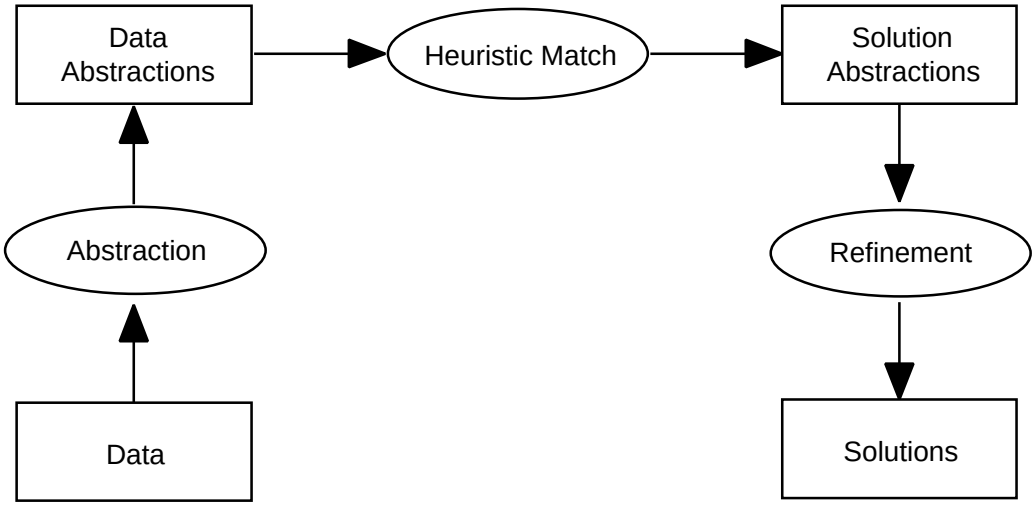

Figure 1. Clancey's heuristic classification model.

The essential role of ontologies and problem solving methods is to support reuse. Hence, not surprisingly, a number of libraries have been developed over the last few years. Some of the libraries focus on problem solving methods - e.g., the Common KADS library (Breuker and Van de Velde, 1994) and Benjamins' library of diagnostic problem solving methods (Benjamins, 1993); some focus on ontologies - e.g., the Stanford Ontolingua repository (Farquhar et al., 1996); some include both ontologies and problem solving methods - e.g.,

Motta's library of parametric design components (Motta, 1999) and the WebOnto repository of knowledge models (Domingue, 1998). The rapid growth in the quantity, quality and size of these repositories attests to the growing importance and maturity of the field. Ontologies are being developed in a variety of domains to support all sorts of computer-based activities, including intelligent internet searches (Guarino et al., 1999; McGuinness, 1998), knowledge acquisition (Motta et al., 2000) and system development (van Heijst, 1995; Motta, 1999), and several success stories have been reported in the literature - see for instance (McGuinness, 1999; Mulholland et al., 2000). Analogously success stories have also been reported about the application of problem solving methods to industrial problems (Speel and Aben, 1997; Hori and Yoshida, 1998). Equally importantly, advances in *knowledge modelling* research (i.e., in research on ontologies and problem solving methods) make it possible to put knowledge system development on a more solid theoretical and engineering basis, thus aligning knowledge engineering with other engineering disciplines - see Schreiber et al. (2000) for an in-depth presentation of a comprehensive framework detailing a methodology and a set of techniques for carrying out structured knowledge analysis and knowledge engineering projects in organizations.

The main goal of the IBROW project is to develop an intelligent brokering service able to retrieve software components from libraries distributed over the Internet and configure them for particular applications in accordance with the stated user requirements. In order to achieve this goal a number of sub-goals need to be achieved. These include: (i) to define an epistemological framework and a formalism that make it possible for users to characterize these components in such a way as to perform semi-automated identification and configuration; (ii) to develop libraries of components according to the aforementioned framework; (iii) to define a brokering agent, both able to reason about the functionalities and costs of available library components and also able to carry out a dialogue with a customer who requires such components; (iv) to define an appropriate interoperability infrastructure which can make it possible to interface, configure and execute the selected components. The first phase of the project has addressed primarily the first goal and has developed a framework, called UPML, which characterizes the types of components which can be found in an IBROW library and their relationships (Fensel et al., 1999a; 1999b). A brief description of UPML is provided in the next section. In this document we focus mainly on the second goal and we present a library of generic, reusable components whose purpose is to support the specification and operationalization of classification problem solvers. In what follows we will discuss our approach to modelling classification problem solving, we will describe the various parts of the library, we will illustrate examples from our application domain and we will discuss related work and open issues. We will start the discussion by illustrating the epistemological framework underlying the library.

2. THE UPML FRAMEWORK FOR REUSE

The basic structure of UPML is illustrated in figure 2. UPML defines four main knowledge components: Tasks, Problem Solving Methods, Domain Models, and Ontologies. A (generic) task specifies a generic class of problems, e.g., parametric design problems (Motta, 1999; Motta et al., 1999; Motta and Zdrahal, 1998). A task specification does not make any assumption about how the problem can be solved, that is, task specifications only define a generic *goal*, not how this can be achieved. A problem solving method provides a (possibly partial) definition of how a generic task can be solved. A domain model specifies a body of knowledge about a particular application domain, with little or no focus on problem solving behaviour. For instance, a domain model could be a database of employees in a company, which describes their role in the organization, their salary, their competences, their performance, etc. Tasks, methods and domain models can be linked to each other by means of *bridges* and specialised by means of *refiners* - see (Fensel et al., 1999b) for more details.

An application can then be developed in a reuse-oriented fashion by selecting an appropriate task model, mapping it to a domain by means of *task-domain bridges*, selecting the appropriate problem solving method and then mapping this to both domain and task. When appropriate, refinements can be used to specialise components according to a particular purpose - e.g., to modify a PSM for a particular domain.

The role of UPML is to provide an architectural framework and a precise characterization of the main types of components, in particular tasks and PSMs, which can be found in libraries of reusable knowledge components. UPML does not impose any commitment with respect to a particular specification language. The idea here is that as long as libraries subscribe to the IBROW architectural framework and components are specified in terms of their functional capabilities, according to the UPML meta-ontology - e.g., PSMs must be characterized in terms of *preconditions*, *domain assumptions* and *postconditions*, then the resulting library will exhibit the correct degree of knowledge-level interoperability, which is required by the IBROW broker.

To carry out the detailed modelling of the classification components we have used the OCML language (Motta, 1999). The advantages of this choice are several: i) our group has extensive experience with this language, having used it for the past 5 years; ii) the language supports both specification and operationalization of knowledge components, thus allowing rapid prototyping and evaluation; iii) we can rely on an extensive library of models, developed in the context of dozens of research projects, which provides an excellent platform for any new modelling initiative; and iv) we have a sophisticated web-based environment, WebOnto (Domingue, 1998), which makes it possible to browse the library from anywhere on the web through a standard web browser, and also supports collaborative model development from teams of developers located at distributed sites. WebOnto is publicly available at http://webonto.open.ac.uk and has now been used in over twenty projects, by research groups from both European and non-European countries. A detailed description of the OCML

language is given in (Motta, 1999); in what follows we will provide some details of OCML constructs, when these are needed to clarify the description of library components.

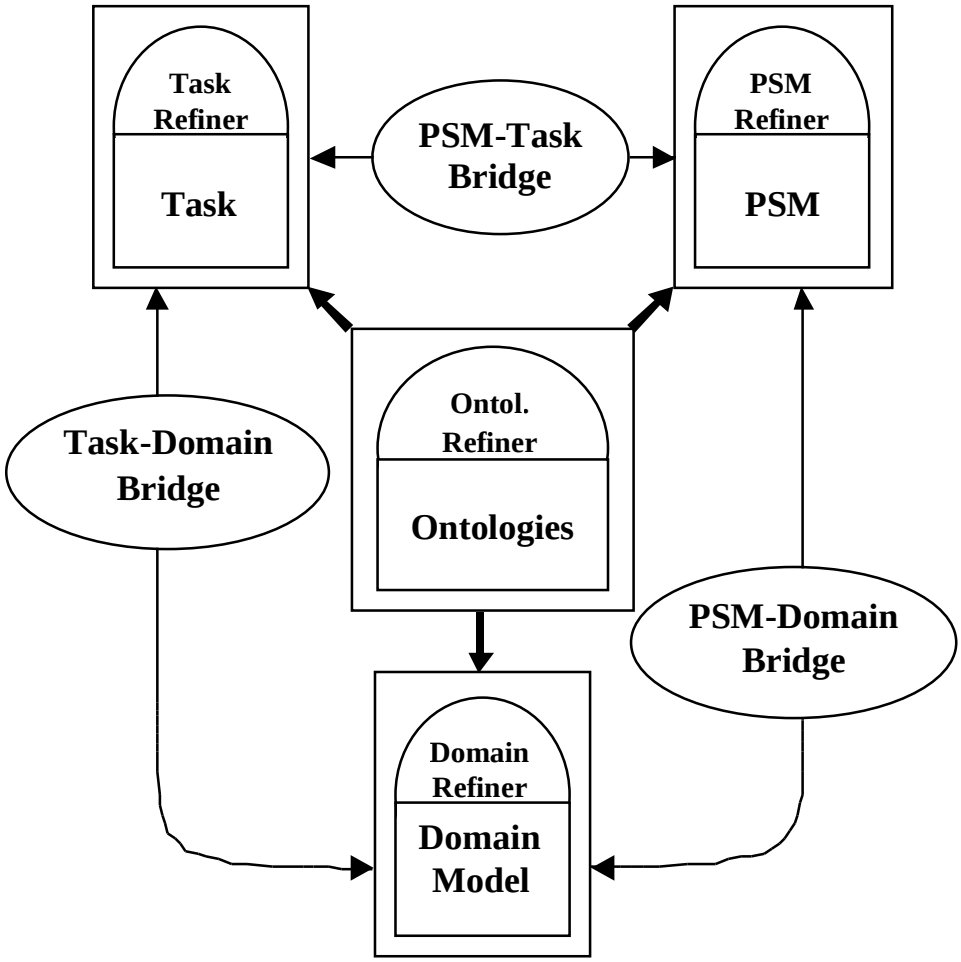

Figure 2. The UPML framework for reuse.

3. MODELLING CLASSIFICATION PROBLEMS

As Stefik (1995) explains, "to classify something... is to identify it as a member of a known class". Hence, classification can be seen as the problem of finding the *solution* (class) which best explains a certain set of known facts (*observables*) about an unknown object, according to some *criterion*. In the following we will discuss each of these concepts in turn.

3.1. Observables

As said above, we use the term 'observables' to refer to the known facts we have about the object (or event, or phenomenon) that we want to classify. Each observable can be characterized as a pair of the form *<f, v>*, where *f* is a *feature* of the unknown object and *v* is its value. Here, we take a very generic viewpoint on the notion of feature. By feature we mean anything which can be used to characterize an object, such that its value can be directly

observed, or derived by inference. As is common when characterizing classification problems - see, e.g., (Wielinga et al., 1998), we assume that each feature of an observable can only have one value. This assumption is only for convenience and does not restrict the scope of the model.

3.2. Solution space

The solution space specifies a set of predefined classes (solutions) under which an unknown object may fall. A solution itself can be described as a finite set of *feature specifications*, which is a pair of the form <*f, c*>, where *f* is a feature and *c* specifies a condition on the values that the feature can take. Then, we can say that an observable *(f, v)* matches a feature specification *(f, c)* if *v* satisfies the condition *c*.

3.3. Criteria for classification tasks

As we have seen, generally speaking, classification can be characterized as the problem of explaining observables in terms of pre-defined solutions. To assess the explanation power of a solution with respect to a set of observables we need to *match* the specification of the observables with that of a solution. Given a solution, *sol*: ((f_{sol1}, c_1).....(f_{solm}, c_m)), and a set of observables, *obs*: ((f_{ob1}, v_1).....(f_{obn}, v_n)), four cases are possible when trying to match them:

- A feature, say f_j, is *inconsistent* if (f_j, v_j) *obs*, (f_j, c_j) *sol* and v_j does not satisfy c_j;
- A feature, say f_j, is *explained* if (f_j, v_j) *obs*, (f_j, c_j) *sol* and v_j satisfies c_j;
- A feature, say f_j, is *unexplained* if (f_j, v_j) *obs* but f_j is not a feature of *sol*.
- A feature, say f_j, is *missing* if (f_j, c_j) *sol* but f_j is not a feature of *obs*.

Given these four cases, it is possible to envisage different solution criteria. For instance, we may accept any solution which explains some data and is not inconsistent with any data. This criterion is called *positive coverage* (Stefik, 1995). Alternatively we may require a *complete coverage* - i.e., a solution is acceptable if and only if it explains all data and is not inconsistent with any data. Thus, the specification of a particular classification task needs to include a *solution (admissibility) criterion*. This in turn relies on a *match criterion*, i.e., a way of measuring the degree of matching between candidate solution and a set of observables. By default, our library provides a match criterion based on the aforementioned model. That is, a *match score* between a solution candidate and a set of observables has the form (*I, E, U, M*), where *I* denotes the set of inconsistent features, *E* the set of explained features, *U* the set of unexplained features and M the set of missing features. Of course users of the library are free to specify and make use of alternative criteria.

In many situations, specifying the conditions under which a candidate solution is indeed a satisfactory solution is not enough. In some cases we may be looking for the best solution, rather than for any admissible one. In this cases we need to have a mechanism for comparing

match scores and this comparison mechanism becomes then part of the specification of the match criterion. By default, our library includes the following *score comparison criterion*.

Given two scores, $S_1 = (i_1, e_1, u_1, m_1)$ and $S_2 = (i_2, e_2, u_2, m_2)$, we say that S_2 is a better score than S_2 if and only if:

$(i_2 < i_1)$

$(i_2 = i_1 \quad e_1 < e_2)$

$(i_2 = i_1 \quad e_2 = e_1 \quad u_2 < u_1)$

$(i_2 = i_1 \quad e_2 = e_1 \quad u_2 = u_1 \quad m_1 < m_2)$

The notation $x_i < x_j$ indicates that the set x_I contains less elements than the set x_j.

The above criterion is of course completely generic. It is possible to envisage domain and/or application-specific comparison criteria that, for instance, give higher importance to some features, rather than others.

In conclusion, our analysis characterizes classification tasks in terms of the following concepts: observables, solutions, match criteria, solution criteria and score comparison criteria. In the next section we illustrate how we have modelled these concepts in OCML.

4. FORMALIZING THE CLASSIFICATION TASK

The ontology for the classification task consists of a group of formal specifications of concepts needed for defining classification tasks. Here we will focus the discussion on the main concepts discussed in the previous section. The complete specification of the ontology can be found in Appendix 1.

4.1. Modelling observables

The definition below defines the notion of observables as an OCML class.

```
(def-class observables (set) ?obs
  "This is simply a set of observables.
   An important constraint is that there cannot be two values for the same
   feature
   in a set of observables"
   :iff-def (every ?obs observable)
   :constraint (not (exists (?ob1 ?ob2)
                            (and (member ?ob1 ?obs)
                                 (member ?ob2 ?obs)
                                 (has-observable-feature ?ob1 ?f)
                                 (has-observable-feature ?ob2 ?f)
                                 (has-observable-value ?ob1 ?v1)
                                 (has-observable-value ?ob2 ?v2)
                                 (not (= ?v1 ?v2))))))
```

The definition states that each element of the class must be a set and that every element of the set must be an observable. In addition it also states that no two members of a set of observables can have the same feature - i.e. the set $((f_1\ v_1)(f_1\ v_2))$ is not a legal set of observables.

The class observable is defined as follows.

```
(def-class observable () ?ob
  "An observable is defined as a pair <feature, value> (logically, not
   necessarily physically).
   This definition assumes that a relation exists, legal-feature-value, which
   specifies whether a value is appropriate for a feature.
   In addition we also assume that only one value can be provided for each
   feature"
  ((has-observable-feature :type feature)
   (has-observable-value :max-cardinality 1))
  :constraint (=> (and (has-observable-feature ?ob ?f)
                       (has-observable-value ?ob ?v))
                  (legal-feature-value ?f ?v)))
```

The definition models an observable in terms of a feature and a value and states that only one value can be specified for each observable (by means of the option `:max-cardinality` in slot `has-observable-value`). The constraint associated with the class states that in an observable the observable value must be legal with respect to the feature. Thus, specific applications and domains can then impose their own legality constraints by providing specific definitions of relation `legal-feature-value`.

Features are simply modelled as classes. Our ontology also includes a subclass of class `feature`, called `contextual-feature`, which can be used to model analyses of classification problems such as the one carried out by Wielinga et al., (1998), which distinguish between *contextual* and *non-contextual* features.

4.2. Modelling solution spaces

A solution space is defined simply as a set of solutions.

```
(def-class solution-space (Set) ?x
  "This is simply a set of solutions"
  :iff-def (every ?x solution))
```

A solution is in turn defined as a set of feature definitions.

```
(def-class solution () ?x
  "A solution is a set of feature definitions"
  :iff-def (every ?x feature-definition))
```

A feature definition is specified in terms of two slots, a feature name and a feature value specification. The latter is a unary relation whose domain is a subset of the domain of allowed values for the feature in question. This requirement is specified by means of the `:constraint` option given in the class definition.

```
(def-class feature-definition () ?x
 "A feature definition is a pair <?f ?spec>, where ?f is a feature and ?spec
  is a unary relation.
  The idea here is that  ?spec defines a sub-domain of ?f, restricting the
  range of values which satisfy the solution"
 ((has-feature-name :type feature)
  (has-feature-value-spec :type unary-relation))
 :constraint (=> (and (has-feature-name ?x ?f)
                      (has-feature-value-spec ?x ?spec))
                 (=> (holds ?spec ?v)
                     (legal-feature-value ?f ?v))))
```

4.3. Modelling match criteria

A match criterion is defined in terms of a scoring mechanism and a match score comparison relation. The former is used to measure the degree of matching between a solution and a set of observables. The latter is used to compare two scores. While we provide a set of definitions specifying these notions generically, the library also provides a specific match criterion, to be used as a default. We will start with the generic definitions.

A match criterion is modelled in terms of a match score mechanism and a score comparison relation.

```
(def-class match-criterion ()
((has-scoring-mechanism :type match-score-mechanism)
 (has-match-score-comparison-relation :type match-score-comparison-relation)))
```

A match score mechanism is modelled as a class `match-score-mechanism` with two slots, one takes instances of the class `macro-score-mechanism` as value and the other takes instances of the class `feature-score-mechanism` as value.

```
(def-class match-score-mechanism ()
  ((has-macro-scoring-mechanism  :type macro-score-mechanism)
   (has-feature-scoring-mechanism  :type feature-score-mechanism)))
```

Both classes `macro-score-mechanism` and `feature-score-mechanism` are subclasses of a class `function`, which is defined in the OCML base ontology. The former specifies the class of functions which take as input a set of observables, a solution and an instance of class `feature-score-mechanism` and return a match score. The latter define the class of functions which take as input a set of observables, a feature, a solution and the list of features associated with the solution and return a *feature-level score*. Therefore a match-score mechanism works at two

levels: one at which each individual <feature value> pair is scored, and the other at which all the scores computed at first level are combined.

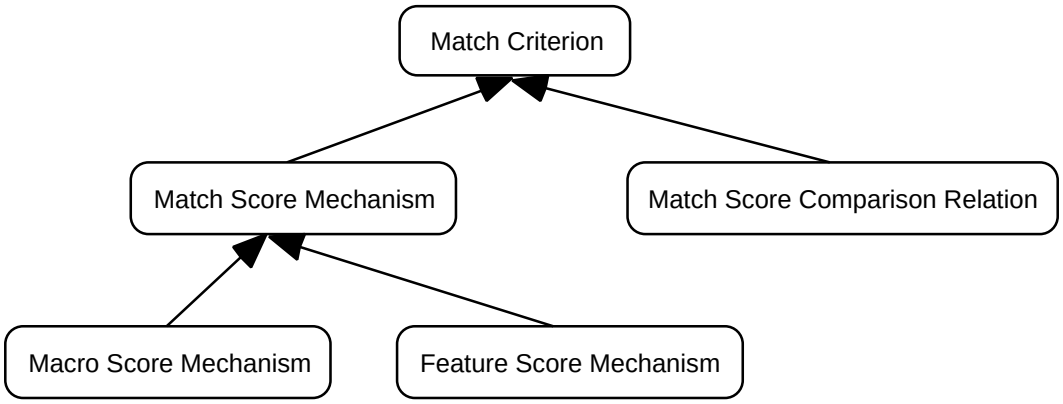

Figure 3. The main classes involved in defining solution space

The score comparison relation is modelled as a binary relation predicated over instances of class `match-score`. We also require a score comparison relation to be a partial order.

```
(def-class match-score-comparison-relation (binary-relation) ?r
  "A match-score-comparison-relation is a binary relation which checks whether
   a score is better than another one"
  :axiom-def (defines-partial-order ?r)
  :constraint (and (domain ?r match-score)
                   (range ?r match-score)))
```

The main classes involved in defining the notion of match criterion are diagrammatically shown in figure 3.

A concrete match criterion can then be defined as an instance of class `match-criterion` - see Appendix 1 for the formal specification of the criterion discussed in section 3.3.

4.4. Solution admissibility criterion

Our class `solution-admissibility-criterion` comprises two slots: one specifies the class of match scores for which this criterion is applicable and the other specifies the unary relation defining the actual admissibility criterion. This relation must be defined over a particular class of match scores. Associating a solution admissibility criterion with a match score results in a cleaner model: whether or not something, say *sol*, qualifies as a solution is a reflection of the results of the matching process between a set of observables and the specification of *sol*.

```
(def-class solution-admissibility-criterion () ?c
  "This provides us with a way to specify when a solution is admissible.
   that is, we can express a threshold under which solutions are not
   feasible.  an admissibility criterion applies to a particular class of match
   scores and is realised through a admissibility relation.  This takes a score
   as input and returns true or false, depending on whether the solution is
   admissible"
 ((applies-to-match-score-type :type match-score-type)
  (has-solution-admissibility-relation :type unary-relation))
 :constraint (=> (and (solution-admissibility-criterion ?c)
                      (has-solution-admissibility-relation ?c ?r)
                      (domain ?r ?d))
                 (subclass-of ?d match-score)))
```

A solution admissibility criterion should be monotonic, that is, if a solution, *sol*, is admissible, then any solution which is better than *sol* must also be admissible. Hence, our ontology includes the following axiom.

```
(def-axiom admissibility-is-monotonic
  "This axiom states that the admissibility  criterion is monotonic. That is,
   if a solution, ?sol, is admissible, then any solution which is better than
   ?sol will also be admissible"
  (forall (?sol1 ?sol2 ?obs ?criterion)
          (=> (and (admissible-solution
                    ?sol1 (apply-match-criterion ?criterion ?obs ?sol1)
                    ?criterion)
                   (better-match-than ?sol2 ?sol1 ?obs ?criterion))
              (admissible-solution
               ?sol2 (apply-match-criterion ?criterion ?obs ?sol2)
               ?criterion))))
```

A specific solution admissibility criterion is defined as an instance of the class `solution-admissibility-criterion`. In our specification we provide two concrete solution admissibility criterions, corresponding to the two criteria discussed in section 3.3. As an illustration we give here the definition of the positive coverage criterion. As usual, more details can be found in Appendix 1.

```
(def-instance complete-coverage-admissibility-criterion
              solution-admissibility-criterion
  ((applies-to-match-score-type default-match-score)
   (has-solution-admissibility-relation
    complete-coverage-admissibility-relation)))

(def-relation complete-coverage-admissibility-relation (?score)
  "Score is a list (I, E, U, M).  According to this criterion a solution should
   be consistent, i.e. I = nil, and explain all features, i.e. U = nil"
  :constraint (default-match-score ?score)
  :iff-def (and (= (length (first ?score)) 0) (= (length (third ?score)) 0)))
```

4.5. Relations ‘best-match’ and ‘admissible’

Solutions can be either optimal or simply admissible. Relation `best-match` is useful to specify which solution is best out of a range of candidates; relation `admissible-solution` defines what is an admissible solution. The two definitions are given below.

```
(def-relation best-match (?obs ?sol ?candidates ?criterion)
  "?class is the best classification for ?obs wrt a set of
   classes if its score - with respect to some criterion - is better than
   that of any other candidate"
  :constraint (member ?sol ?candidates)
  :iff-def (and (member ?sol ?candidates)
                (sol-has-match-score ?sol ?obs ?score ?criterion)
                (not (exists ?sol2
                             (and (member ?sol2 ?candidates)
                                  (not (= ?sol2 ?sol))
                                  (sol-has-match-score ?sol2 ?obs ?score2
                                                       ?criterion)
                                  (better-match-score ?score2 ?score
                                                      ?criterion))))))

(def-relation admissible-solution (?sol ?score ?criterion)
  :iff-def (holds (the ?rel (has-solution-admissibility-relation ?criterion
                                                                 ?rel))
                  ?score))
```

Relation `best-match` states that a solution, *sol*, is the best with respect to a criterion and a set of observables if its score is better than that of anybody else in the given set of candidates. A solution *sol* with a certain score is admissible with respect to a solution admissibility criterion if the admissibility relation associated with the criterion holds for the given score.

4.6. Auxiliary relations and functions

The definition of the classification ontology involves a lot of auxiliary functions and relations, many of which have been used in the examples given in this section. For convenience we list some of the most significant here. The complete ontology contains 42 definitions.

Relations

better-match-than(?sol1, ?sol2, ?obs, ?mc)	True if for the observables ?*obs* and match criterion ?*mc*, *?sol1* has better match score than *sol2.*
sol-has-match-score(?sol,?obs, ?score, ?mc)	True if for observables ?*obs* and match criterion ?*mc*, ?*sol* has match score ?*score*.
better-match-score(?score1, ?score2, ?mc)	True if ?*score1* is a better score than ?*score2* according to match criterion ?*mc*
composable-solutions (?sol1 ?sol2)	Two solutions are composable if they are not mutually inconsistent with respect to some feature.

Functions

apply-match-criterion (?mc ?obs ?sol)	Return the score of ?*sol* for the observables ?*obs* and match criterion ?*mc*
the-better-match-score (?d1 ?d2 ?mc)	Return the better score according to match criterion ?*mc*
all-features-in-observables (?obs)	This function extracts the features from a set of observables
observables-feature-value (?obs ?f)	This function retrieves the value of a feature in a set of observables. The assumption is that a feature can at most have one value in a set of observables. If ?f does not appear in ?obs, then :nothing is returned
all-features-in-solution (?sol)	"This returns all the features in a solution
solution-feature-spec (?sol ?f)	"This returns the spec associated with a particular feature in a solution"

5. CLASSIFICATION TASK SPECIFICATION

Having defined the task ontology, we are now in a position to present a formal specification of a family of classification tasks. In UPML a task is specified in terms of its *input* and *output roles*, *preconditions*, *assumptions* and *goal*. The same approach is taken in the OCML *base ontology*, which formally defines the notion of task used in OCML. In particular, the OCML base ontology defines a goal as a *kappa expression* (i.e., an anonymous relation) with two arguments: the task itself and a value. The latter is meant to be a possible result from carrying out the task. The goal of a task, say *task*, is satisfied for a value *v*, if the associated kappa expression holds for arguments *task* and *v*.

Several definitions of classification tasks can be provided. In some cases we may be asking for optimal solutions, in other cases we just want to find an admissible one. Hence, here we provide a family of task specifications. We start with a generic definition, which defines the input and output roles of classification tasks, the relevant preconditions and a default goal specification. Different variants on the classification task can then be easily defined as refinements of this generic definition.

```
(def-class classification-task (goal-specification-task) ?task
  ((has-input-role :value has-candidate-solutions
                           :value has-observables
                           :value has-match-criterion
                           :value has-solution-admissibility-criterion)
   (has-output-role :value has-solutions)
   (has-candidate-solutions :type solution-space)
   (has-observables :type observables)
   (has-match-criterion :type match-criterion
                        :default-value default-match-criterion)
   (has-solution-admissibility-criterion
    :type solution-admissibility-criterion
    :default-value default-solution-admissibility-criterion)
   (has-solutions :type solution-space)
   (has-precondition
    :value (kappa (?task)
                  ?x  ?y (member  ?x ?task.has-observables)
                         (member ?y ?task.has-candidate-solutions))
   (has-goal-expression
    :documentation "The goal is to find one or more admissible solutions"
    :default-value (kappa (?task ?sols)
                     ?sol (member ?sol ?sols) →
                               (admissible-solution
                                ?sol (apply-match-criterion
                                      ?task.has-match-criterion
                                      ?task.has-observables
                                      ?sol)
                                ?task.has-solution-admissibility-criterion))
```

The above definition specifies four input roles:

- `has-candidate-solutions` whose filler must be an instance of class `solution-space`.
- `has-observables` whose filler must be an instance of class `observables`.
- `has-match-criterion` whose filler must be an instance of class `match-criterion`. A default value is also provided for this role, `default-match-criterion`.
- `has-solution-admissibility-criterion` whose filler must be an instance of class `solution-admissibility-criterion`. A default value is also provided for this role, `default-solution-admissibility-criterion`.

The output role is the following:

- `has-solutions` whose filler must be an instance of class `solution-space`.

The task precondition states that a classification task is meaningful if and only if some observables and some candidate solutions are provided. The (default) goal expression states that each solution output must be admissible with respect to the given solution admissibility criterion.

In many cases we know that only one solution to a classification task exists (or alternatively we may not care about retrieving multiple solutions, one will do). We can model this situation

by refining the above definition and introducing an assumption which states that the output of the task will include at most one solution. This assumption can then be used by the IBROW broker when choosing an appropriate problem solving method and when instantiating the task for a particular domain. A single-solution classification task can then be formalised as follows:

```
(def-class single-solution-classification-task (classification-task) ?task
  "A classification task which assumes only one solution is found.  The
   assumption just states that the output of the task will contain at most one
   solution. It does not say anything about whether only one solution actually
   exists."
  ((has-assumption
    :value (kappa (?task)
                  (< (Length ?task.has-solutions) 2)))))
```

Finally we give the definition of a classification task which requires a solution to be optimal with respect to the given match criterion.

```
(def-class optimal-classification-task (classification-task) ?task
 ((has-goal-expression
       :default-value (kappa (?task ?sols)
                      ?sol (member ?sol  ?sols) →
                               ((admissible-solution
                                  ?sol
                                  (apply-match-criterion
                                   ?task.has-match-criterion
                                   ?task.has-observables
                                  ?sol)
                                  ?task.has-solution-admissibility-criterion)

                                 (best-match ?task.has-observables
                                             ?sol
                                             ?task.has-candidate-solutions)
                                             ?task.has-match-criterion))))))
```

6. PROBLEM SOLVING METHODS FOR CLASSIFICATION

6.1. Classification problem solving

Classification involves searching the space of possible solutions to find one which explains the given observables in accordance with the given solution criterion. The choice of a method depends on many factors, such as whether the goal is to find one, all or the best solution; whether all observables are known at the beginning or are uncovered opportunistically (maybe at some cost) during the problem solving process; whether or not the solution space is structured according to a *refinement hierarchy*; whether solutions can be composed together, or alternatively, whether each solution presents a different, self-contained alternative. The current version of the library provides support to deal with most of these scenarios. However

it currently does not offer any method to acquire observables incrementally, nor it supports composite solutions. It is envisaged that support for these problem solving approaches will be added to future versions of the library.

If we restrict ourselves to a scenario in which observables are not acquired incrementally during the classification process, then only a *data-directed* approach can be taken. That is, from the given set of observables we identify the most promising solutions and, in those cases where a refinement hierarchy is provided, we can then search the latter to home in on the most appropriate solutions. A data-directed approach does not however imply that the set of observables remain the same during the problem solving process. In many cases we can have *abstraction mechanisms*, which generate new observables from existing ones, in accordance with Clancey's model shown in figure 1. From this discussion it follows that we can use the heuristic classification model as the basis of our library of problem solving methods for classification. Simpler problem solving methods can simply be defined as 'degenerations' of the heuristic classification model – e.g., these may not include abstraction and/or refinement mechanisms. More complex PSMs – e.g., PSMs which allow for composite solutions to be generated, will instead extend this model with additional inference mechanisms.

As we did in the previous section we will present our PSM specification in two steps: first we define a method ontology, which formalises the necessary concepts, relations and functions. Then we will define the actual PSMs.

6.2. A method ontology for classification problem solving

The heuristic classification model includes two notions not already covered by our classification task ontology: abstractors and refiners. These are defined below.

6.2.1. Abstractors

Data abstraction consists of inferring new observables from existing ones by means of domain knowledge. We model this kind of knowledge by means of *abstractors*. Formally, an abstractor is a function which takes a set of observables as input and produces an observable as output. Hence, we can model it as a subclass of class `function` with an additional slot, called `applicability-condition`, whose filler must be an instance of class `abstractor-applicability-condition-class`. This slot can be used to specify precisely additional domain-specific conditions associated with a particular class of abstractors.

```
(def-class abstractor (function)
 ((domain :value observables)
  (range :value observable)
  (applicability-condition :type abstractor-applicability-condition-class)))
```

The applicability condition for an abstractor is defined as a relation whose domain is a subset of class `observables`.

```
(def-class abstractor-applicability-condition-class (unary-relation) ?x
   :iff-def (subset (the ?d (domain ?x ?d)) observables))
```

Abstractors implicitly introduce a hierarchy structure in the data space: directly observed data are at the lowest level of the hierarchy, higher level data are obtained by applying abstractors to lower level data.

6.2.2. Refiners

Solution refinement consists of taking a solution as input and generating more specialised ones – i.e., solutions which cover a subset of the phenomena explained but the 'parent' solution. In this process, domain knowledge about the solution is used. We model this kind of knowledge by means of *refiners*. Refinement can be seen as the inverse process to abstraction, rather than going from concrete data to abstract ones, here we go from abstract solutions to more concrete ones. Hence, not surprisingly we model the class of refiners much the same way as we modelled the class of abstractors, as a subclass of class `function` augmented with a slot which can be used to specify additional, domain-dependent applicability conditions.

```
(def-class refiner (function)
 ((domain :value solution)
  (range :value solution-space)
  (applicability-condition :type refiner-applicability-condition-class)))
```

The applicability condition for a refiner must be a relation and its domain must be a subset of class solution.

```
(def-class refiner-applicability-condition-class (unary-relation) ?r
  :iff-def (subset (the ?x (domain ?r ?x)) solution))
```

Refiners also introduce a hierarchy structure in the solution space: the level of the solution appearing in the slot *domain* of a refiner can be viewed as one level higher than that of the solutions appearing in the slot *range* of the refiner.

A typical case is that in which solutions are represented as classes and the class/subclass hierarchy is used to represent the space of refinements. To cover this obvious scenario our library provides the following refiner instance, which returns all direct subclasses of the input class (i.e., solution).

```
(def-instance refinement-through-subclass-of-links refiner
 "If the solution space is specified by means of classes arranged in a
 subclass-of hierarchy, then this is a good refiner to use"
 ((has-body '(lambda (?sol)
               (setofall ?sub (direct-subclass-of ?sub ?sol))))
  (applicability-condition ‘(kappa (?sol)
                                   (and (class ?sol)
                                        (exists ?sub
                                          (direct-subclass-of
                                           ?sub ?sol)))))))
```

6.2.3. Candidate exclusion criterion

As pointed out earlier, classification problem solving can be seen as the process of searching a solution space, in accordance with the given application criteria. Loosely speaking efficient search requires the ability to identify promising paths and the ability to stay away from dead ends. Our task ontology already provides generic components which make it possible for developers to represent ‘positive notions’ such as when a solution is admissible. Here, we provide ontological support for specifying ‘negative notions’, such as a criterion to be used to decide when a candidate solution and its refinements can be removed from the solution search space. We call this the *candidate exclusion criterion*. As in the case of the solution admissibility criterion, we model it as a class with two slots, which specify the relevant match score type and the actual relation used to test whether a solution should be ruled out.

```
(def-class candidate-exclusion-criterion ()
   ((applies-to-match-score-type :type match-score-type)
    (has-candidate-exclusion-relation :type unary-relation) ))
```

A correct candidate exclusion criterion is monotonic in the sense that if a solution, *sol*, is ruled out, then any solution which has a worse score than *sol* will also be ruled out. Hence, our ontology includes the following axiom.

```
(def-axiom exclusion-is-monotonic
     ?sol1  ?sol2  ?obs  ?criterion
    (ruled-out-solution ?sol1 (apply-match-criterion ?criterion ?obs ?sol1)
                        ?criterion)
    (¬(better-match-than ?sol2 ?sol1 ?obs ?criterion))
  →
  (ruled-out-solution ?sol2 (apply-match-criterion ?criterion ?obs  ?sol2)
                      ?criterion)
```

In addition we also require that the admissibility and exclusion criteria associated with the same match criterion have to be congruent: no solution should ever satisfy both. Formally we have:

```
(def-axiom congruent-admissibility-and-exclusion-criteria
   ?sol  ?task
    (member ?sol (the-virtual-solution-space
                  ?task.has-candidate-solutions
                  ?task.has-refiners))
   →
     ¬ ((admissible-solution ?sol (apply-match-criterion
                                    ?task.has-match-criterion
                                    ?task.has-observables
                                    ?sol)
                               ?task.has-solution-admissibility-criterion)

         (ruled-out-solution ?sol (apply-match-criterion
                                    ?task.has-match-criterion
                                    ?task.has-observables
                                    ?sol)
                               ?task.has-solution-exclusion-criterion)))
```

Our library provides a default solution exclusion criterion which rules out inconsistent solutions – see definitions below.

```
(def-instance default-solution-exclusion-criterion solution-exclusion-criterion
  ((applies-to-match-score-type default-match-score)
    (has-candidate-exclusion-relation default-candidate-exclusion-relation)))

 (def-relation default-candidate-exclusion-relation (?score)
   :constraint (default-match-score ?score)
   :iff-def (> (length (first ?score)) 0))
```

6.2.4. Auxiliary relations and functions

Our method ontology for classification problem solving includes many auxiliary relations and functions. Below we list some of the most significant.

Functions

the-virtual-solution-space (?init-space ?refs)	Return the solution space generated by refinement application from an initial solution space
apply-abstract-operator (?ab ?obs--in)	Return a new observable obtained by applying abstractor ?*ab* to observables ?*obs-in.*
apply-refiner-operator (?ref ?solution)s	Return refined solutions obtained by applying *?ref* to observables ?*obs-in.*

Relations

ruled-out-solution (?sol ?score ?sec)	True if the ?*score* of the solution *?sol* satisfies the relation defined in the solution exclusion criterion *?sec*.
abstractor-is-applicable? (?ab ?obs)	True if abstractor *?ab* can be applied to observables *?obs*
generalized-abstract-from (?obs-out ?obs-in ?abs)	True if *?obs-out* is obtained by applying abstractors *?abs* to observables *?obs-in*.
generalised-abstract-link (?ob1 ?ob2 ?abs)	?*ob1* is in a chain of abstraction which stems from ?*ob2*
observable-abstracted-from (?ob ?obs ?ab)	True if observable *?ob* is abstracted from observables *?obs* using abstractors in *?abs*
refiner-is-applicable? (?ref ?sol)	True if abstractor *?ref* can be applied to solution *?sol*
some-refiner-is-applicable? (?refs ?sol)	True if there is a refiner in *?refs* that is applicable to *?sol*
generalised-refinement-of (?sols-out ?sols-in ?refs)	True if *?sols-out* is obtained by applying refiners *?refs* to solutions *?sols-in*.

6.3. Specification of heuristic classification problem solving methods

A problem solving method in OCML is defined as a subclass of a predefined class `problem-solving-method`, whose main slots are as follows:

- `has-input-role`: all inputs to the method are listed here;
- `has-output-role`: the output produced by the method;
- `has-precondition`: the preconditions associated with the method;
- `has-postcondition`: the postconditions associated with the method;
- `has-assumption`: the domain assumptions imposed by the method;
- `has-body`: the control structure of the method.

Basically, the pre- and post-conditions together describe the competence of the method. The slot `has-body` specifies its procedural behaviour. In general the same competence can be achieved by different procedural behaviours. The slot `has-assumption` specifies the conditions which have to be satisfied by the domain knowledge to ensure that the specified behaviour can deliver the method's competence for a particular domain.

Our library comprises two basic generic methods, one for dealing with classification tasks where optimal solutions are not required, the other for dealing with classification tasks where optimal solutions are sought. We will first describe the former.

6.3.1. PSM heuristic-admissible-sol-classifier

This is a basic method for finding a single admissible solution to a classification problem. More precisely it can tackle instances of class `single-solution-classification-task`. The single solution assumption greatly reduces the complexity of the search. This method uses a backtracking hill-climbing solution to home in on the most promising solutions. It is a heuristic, hierarchical classifier, as it provides for both abstraction and refinement mechanisms. These are however are optional. In particular, if no refinement mechanisms are provided, then the psm just reduces to a 'flat' classifier.

For the sake of explanation, we will discuss the various parts of the specification separately.

```
(def-class heuristic-admissible-sol-classifier (problem-solving-method) ?psm
  ((has-input-role :value has-abstractors
                   :value has-refiners
                   :value has-observables
                   :value has-candidate-solutions
                   :value has-solution-exclusion-criterion)
   (has-output-role :value has-solution)
   (has-abstractors :type abstractors)
   (has-refiners :type refiners)
   (has-observables :type observables)
   (has-solution-exclusion-criterion
    :type candidate-exclusion-criterion
    :default-value default-candidate-exclusion-criterion)
   (has-candidate-solutions :type solution-space)
   (has-solution :type solution)
.............................
   (has-output-mapping
    :value '(lambda (?psm ?result) (list-of ?result)))

  :own-slots ((tackles-task-type single-solution-classification-task)))
```

The above definition shows the role definition part of the method specification. In addition to the input roles inherited from the definition of the classification task, this method also uses knowledge about abstraction and refinement and a candidate exclusion criterion. None of these types of knowledge actually need to be provided by the user. By default, the method will assume that no refinement or abstraction mechanisms exist and it will use the default candidate exclusion criterion described in the previous section.

The definition also points out that the PSM has been defined to tackle single-solution classification tasks and that the output from the PSM needs to be modified, in order to conform to the list format expected from classification tasks - this is specified by means of slot `has-output-mapping`. This facility provides a simple adaptation mechanism for adapting PSMs to tasks - see section 9.4 for a discussion of the extent to which the OCML adaptation facilities support the UPML framework.

The method also includes four assumptions. The first two state that the abstraction and refinement hierarchies have to be free of cycles.

```
(kappa (?psm)
        (not (exists (?ob1 ?ob2)
                     (and (or (member ?ob1 ?psm.has-observables)
                              (exists ?ob
                                      (and (member ?ob ?psm.has-observables)
                                           (generalised-abstract-link
                                            ?ob1 ?ob ?psm.has-abstractors))))
                          (generalised-abstract-link
                           ?ob2 ?ob1 ?psm.has-abstractors)
                          (generalised-abstract-link
                           ?ob1 ?ob2 ?psm.has-abstractors)))))

(kappa (?psm)
       (not (exists (?sol1 ?sol2)
            (and (or (member ?sol1 ?psm.has-candidate-solutions)
                     (exists ?sol (and (member
                                        ?sol ?psm.has-candidate-solutions))
                                       (generalised-refinement-of
                                        ?sol1 ?sol ?psm.has-refiners))))
                 (generalised-refinement-of
                  ?sol2 ?sol1 ?psm.has-refiners)
                 (generalised-refinement-of
                  ?sol1 ?sol2 ?psm.has-refiners))))
```

The third assumption states that the PSM assumes the existence of a solution in the virtual solution space - i.e., if such a solution exists, the PSM will find it.

```
(kappa (?psm)
       (exists ?sol
               (and (admissible-solution
                     ?sol
                     (apply-match-criterion ?psm.has-match-criterion
                                            ?psm.has-observables
                                            ?sol)
                     ?psm.has-solution-admissibility-criterion)
                    (member ?sol
                            (the-virtual-solution-space
                             ?psm.has-candidate-solutions
                             ?psm.has-refiners)))))
```

The final assumption states that the method assumes that the exclusion criterion is monotonic with respect to the refinement hierarchy. That is, it assumes that if a solution is excluded, all its refinements can be excluded too.

```
(kappa (?psm)
       (forall (?sol ?score)
               (=>
                (and
                 (sol-has-match-score ?sol
                                      ?psm.has-observables
                                      ?score
                                      ?psm.has-match-criterion)
                 (ruled-out-solution ?sol ?score
                                     ?psm.has-solution-exclusion-criterion))
                (not (exists
                      ?sol2
                      (and (generalised-refinement-of
                            ?sol2 ?sol ?psm.has-refiners)
                           (admissible-solution
                            ?sol2
                            (apply-match-criterion ?psm.has-match-criterion
                                                   ?psm.has-observables
                                                   ?sol2)
                            ?psm.has-solution-admissibility-criterion)))))))
```

The postcondition of the method states that the output solution satisfies the admissibility criterion - i.e., assuming the domain assumptions are satisfied, the method will return an admissible solution.

```
(has-postcondition
 :value (kappa (?psm ?sol)
               (admissible-solution
                ?sol
                (apply-match-criterion ?psm.has-match-criterion
                                       ?psm.has-observables
                                       ?sol)
                ?psm.has-solution-admissibility-criterion)))
```

The body of the PSM is very simple - see definition below. It calls the abstraction task, to apply all available abstractors and then calls a search procedure which is able to search the solution space for an admissible solution.

```
begin
  ?obs := (achieve-generic-subtask ?psm abstraction
                                   has-observables ?psm.has-observables
                                   has-abstractors ?psm.has-abstractors)
  (admissible-solution-search ?psm ?obs ?psm.has-refiners
                              ?psm.has-candidate-solutions
                              ?psm.has-solution-match-criterion
                              ?psm.has-solution-admissibility-criterion
                              ?psm.has-solution-exclusion-criterion)
end
```

The admissible-solution-search procedure works as follows. If one of the candidate solution given as input is an admissible solution (according to the given criterion), then the procedure

stops and return the solution it has found. Otherwise it ranks the candidates according to the given match criterion (recall that a match criterion also includes a score comparison relation) and then explore the refinement space of each one in turn, until it finds a solution. This behaviour is analogous to hill climbing with backtracking. At each stage of the search process the most promising path is explored, but if it is found to be a dead end then the procedure is able to backtrack to the next one in the most recent branching point.

```
(def-procedure admissible-solution-search (?psm ?obs ?refs ?candidates
                                           ?m-criterion ?a-criterion
                                           ?e-criterion)
 :body
   begin
    if    ?sol ((member ?sol  ?candidates)
                (admissible-solution ?sol (apply-match-criterion
                                           ?m-criterion ?obs ?sol)
                                     ?a-criterion))
     then return ?sol
     else
      begin
       ?ranked-candidates := (achieve-generic-subtask ?psm rank-solutions
                              has-observables ?obs
                              has-candidate-solutions ?current-sols
                              has-match-criterion ?m-criteria)
            loop for ?candidate in ?ranked-candidates do
             if  ¬ (ruled-out-solution ?candidate
                                       (apply-match-criterion ?m-criteria
                                                              ?obs
                                                              ?candidate)
                                       ?e-criteria)
               then
                 begin ?refined-sols := (achieve-generic-subtask
                                         ?psm refinement
                                         has-candidate-solution ?candidate
                                         has-observables ?obs
                                         has-refiners  ?refs)
                      if  ?refined-sols ≠ nil then
                         begin
                           ?result := (admissible-solution-search
                                       ?psm ?obs ?refs ?refined-sols
                                       ?m-criterion ?a-criterion e-criterion)
                                if ?result ≠ :nothing  then return ?result
                           end if
                         end
                   end if
                 end
             end if
           end loop
      end
    end if
 end
```

6.3.2. PSM heuristic-optimal-sol-classifier

This PSM can be used to tackle classification tasks which require a solution to be optimal with respect to the given match criterion. The method carries out an exhaustive search looking for an optimal solution and uses the exclusion criterion to prune the search space. It returns one or more optimal solutions.

This method has the same input role specification and assumptions as the PSM described in the previous section, therefore we only need to describe its postcondition and its control regime. Its postcondition is defined as follows.

```
(kappa (?psm ?sols)
       (forall ?sol
               (=> (member ?sol ?sols)
                   (and

                    (admissible-solution
                     ?sol
                     (apply-match-criterion ?psm.has-match-criterion
                                            ?psm.has-observables)
                                            ?sol)
                     ?psm.has-solution-admissibility-criterion)

                    (best-match
                     ?psm.has-observables
                     ?sol
                     (the-virtual-solution-space
                      ?psm.has-observables
                      ?psm.has-candidate-solutions)
                      ?psm.has-match-criterion)))))
```

The definition given above states that any solution returned by the optimal classification method will be both admissible and optimal with respect to the given admissibility and match criteria.

The next definition gives the control regime of the optimal classification method. The method works as follows. It first performs data abstraction and ranks the input candidates in terms of the given match criterion. Then it initializes the current solutions, by checking whether the best ranked candidate, say *sol*, is already an admissible solution. If this is the case, then the PSM makes use of the monotonicity property of solution admissibility criteria to add to the list of current solutions all the given candidates whose score is not worse than *sol*. Then the method checks whether any candidate can be ruled out. It does so by going through the ranked candidates in reverse order (from the worst to the best) and stopping as soon as it finds a candidate which cannot be ruled out. Here the PSM makes use of the monotonicity property of solution exclusion criteria: if a candidate sol_1 cannot be ruled out, then a candidate sol_2 which is better than sol_1 cannot be ruled out either. Once the set of current solutions and ranked candidates have been initialized, then the PSM invokes procedure `complete-optimal-search` to perform a complete search of the solution space.

```
begin
  ?obs :=  (achieve-generic-subtask
             ?psm abstraction has-observables ?psm.has-observables
                              has-abstractors ?psm.has-abstractors)
  ?psm.has-ranked-candidates := (achieve-generic-subtask
                                 ?psm rank-solutions
                                  has-observables ?obs
                                  has-candidate-solutions
                                  ?psm.has-candidate-solutions
                                  has-match-criterion ?psm.has-match-criterion)
  ;;Initialize current solutions
  ?psm.has-current-solutions := nil
  if (admissible-solution (first ?psm.has-ranked-candidates)
                          (apply-match-criterion
                             ?psm.has-match-criterion ?obs
                             (first ?psm.has-ranked-candidates)
                        ?psm.has-solution-admissibility-criterion)
    then
     begin ?psm.has-current-solutions := (List-of
                                          (first ?psm.has-ranked-candidates))
           loop for ?candidate in (rest ?psm.has-ranked-candidates)
              do
             if (better-match-than (first ?psm.has-ranked-candidates)
                                   ?candidate ?obs
                                   ?psm.has-match-criterion)
                 then return :nothing
                 else
                   ?psm.has-current-solutions := (cons
                                                  ?candidate
                                                  ?psm.has-current-solutions)
              end if
            endloop
     end
  end if

  ;;We now remove ruled out candidates.
  loop for ?candidate in (reverse ?psm.has-ranked-candidates)
        do
         if (ruled-out-solution ?candidate (apply-match-criterion
                                            ?psm.has-match-criterion
                                            ?obs ?candidate)
                                           ?psm.has-exclusion-criterion)
             then ?psm.has-ranked-candidates := (remove ?candidate
                                                        ?psm.has-ranked-candidates)
              else return :nothing
         end if
  endloop

  if ?psm.has-ranked-candidates = nil
   then ?psm.has-current-solutions
   else (complete-optimal-search ?psm ?obs ?psm.has-refiners
          (first ?psm.has-ranked-candidates)
          (rest ?psm.has-ranked-candidates) ?psm.has-match-criterion
          ?psm.has-solution-admissibility-criterion
          ?psm.has-solution-exclusion-criterion
  end if
end
```

```
(def-procedure complete-optimal-search (?psm ?obs ?refs ?candidate
                                        ?other-candidates ?current-solutions
                                        ?match-criterion ?admissibility-criterion
                                        ?exclusion-criterion)
  begin
    ?good-refined-sols := (filter (achieve-generic-subtask ?psm refinement
                                   has-candidate-solution ?candidate
                                   has-observables ?obs has-refiners ?refs)
                                  (kappa (?sol)
                                   ¬(ruled-out-solution ?sol
                                     (apply-match-criterion
                                       ?match-criterion ?obs ?sol)
                                     ?exclusion-criterion))))
    if ?good-refined-sols = nil
     then if ?other-candidates = nil
            then ?current-solutions
            else (complete-optimal-search
                   ?psm ?obs ?refs (first ?other-candidates)
                   (rest ?other-candidates) ?current-solutions ?match-criterion
                   ?admissibility-criterion ?exclusion-criterion))
          end if
     else
     begin   ;;there are some new useful refinements
       ?ranked-candidates := (achieve-generic-subtask
                              ?psm rank-solutions
                              has-observables ?obs
                              has-candidate-solutions
                                 (append ?other-candidates ?good-refined-sols)
                              has-match-criterion  ?match-criterion)
       ?new-admissible-solutions := (filter ?good-refined-sols
                                            '(kappa (?sol)
                                               (admissible-solution
                                                ?sol
                                                (apply-match-criterion
                                                  ?match-criterion ?obs ?sol)))
                                                ?admissibility-criterion)
       if ?new-admissible-solutions = nil
        then ?new-current-solutions := ?current-solutions
        else ?ranked-sols := (achieve-generic-subtask
                              ?psm rank-solutions has-observables ?obs
                              has-candidate-solutions
                               (append ?current-solutions
                                       ?new-admissible-solutions)
                              has-match-criterion ?match-criterion)
             ?new-current-solutions :=  (cons (first ?ranked-sols)
                                              (filter (rest ?ranked-sols)
                                                      '(kappa (?sol)
                                                        (¬ (better-match-than
                                                            (first ?ranked-sols)
                                                            ?sol ?obs
                                                            ?match-criterion)))))))
       end if
       (complete-optimal-search ?psm ?obs ?refs (first ?ranked-candidates)
                                (rest ?ranked-candidates)
                                ?new-current-solutions ?match-criterion
                                ?admissibility-criterion ?exclusion-criterion)
end
```

If no candidates exist (maybe because they have been ruled out by the solution exclusion criterion), then the method returns the set of current solutions.

This search is implemented by the search procedure *complete-optimal-search.* This takes as input a PSM, a set of observables, a set of refiners, the current best candidate, a ranked list of other possible candidate solutions, the list of currently best solutions and the match, admissibility and exclusion criteria. Its output is the list of best solutions generated after exhaustive search of the refinement space. The procedure assumes that both the current best candidate and any other candidate in the `other-candidates` list are possible solutions - i.e., they have passed the exclusion criterion test.

The procedure works as follows. First, it calls task `refinement` to generate the refinements of the current candidate solution and it uses the solution exclusion criterion to filter out the 'bad' candidates. If no good refinements have been found and there are no other candidates, then the current set of solutions is returned. Otherwise, if some other candidate can be tried, a recursive call of *complete-optimal-search* is invoked. The only case left is the one in which some 'good' refinements have been generated from the current candidate. In this scenario we do the following: i) we rank the list obtained my merging the current set of candidates with the new refined solutions according to the match criterion and ii) we check whether any of the refined solutions is an admissible solution to our current classification problem. If there are new solutions, then we check whether any of these is better than our current set of solutions (remember that we are looking for optimal solutions) and we update the list of optimal solutions accordingly. Then we recursive invoke *complete-optimal-search* with the updated set of current solutions, the current top candidate and the other candidates in order of ranking. Thus, in accordance with the *best-first* search strategy (Stefik, 1995), this PSM always explores the best node currently known, rather than performing only local backtracking steps.

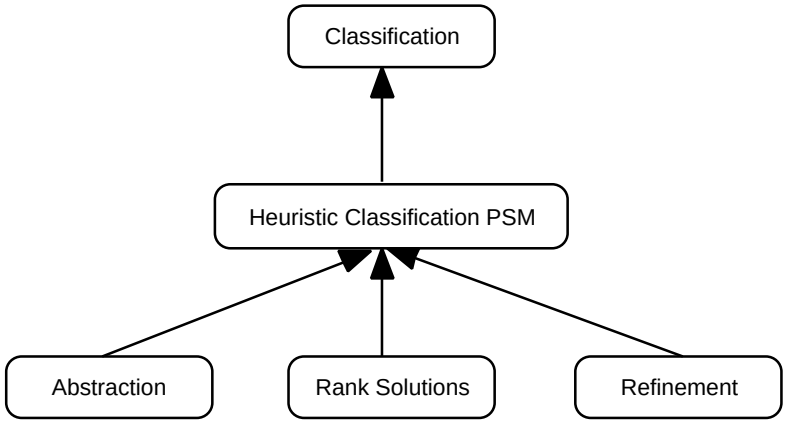

Figure 4. Main tasks for classification problem solving.

6.4. Subtasks

The two PSMs we have introduced for classification tasks share the same input specification, assumptions and subtasks. Essentially, they provide different ways of searching a solution

space. Figure 4 shows the direct subtasks of these PSMs: abstraction, refinement and solution ranking. In this section we present the specifications of each of these subtasks in turn.

6.4.1. Abstraction task and abstraction PSM

The goal of the abstraction task is to abstract from the given observables, using the given abstraction mechanisms. The output includes both the original (non-abstracted) observables and those inferred by means of abstraction mechanisms. The goal expression states also that the available abstraction mechanisms should be applied exhaustively.

```
(def-class abstraction (goal-specification-task) ?task
  "The goal of this task is to abstract from the given observables, using the given
   abstractors.  Note that the output also includes the original (non-abstracted)
   observables"
  ((has-input-role :value has-observables
                   :value has-abstractors)
   (has-output-role :value has-abstract-observables)
   (has-observables :type observables)
   (has-abstract-observables :type observables)
   (has-abstractors :type list)
   (has-goal-expression
    :value (kappa (?task ?observables)
                  (and (forall ?ob
                               (=> (member ?ob ?observables)
                                   (or (member ?ob ?task.has-observables)
                                       (generalized-abstract-from
                                        ?observables
                                        ?task.has-observables
                                        ?task.has-abstractors))))
                       (not (exists (?ab ?new-ob)
                                    (and (member ?ab ?task.has-abstractors)
                                         (abstractor-is-applicable?
                                          ?ab ?observables)
                                         (= ?new-ob (apply-abstract-operator
                                                     ?ab ?observables))
                                         (not (member ?new-ob
                                                      ?observables)))))))))))
```

Our classification library contains an ‘obvious’ method which applies abstractors to the given data space (observables) repeatedly until no more abstractions can be performed. It returns the complete set of observables - i.e., both those inferred by abstraction and those given as input.

```
(def-class abstraction-psm (primitive-method) ?psm
  ((has-control-role :value has-current-observables
                     :value has-current-abstractors)
   (has-current-observables :type observables)
   (has-current-abstractors :type list)
   (has-body
    :value  '(lambda (?psm)
               (do
                 (set-role-value ?psm 'has-current-observables
                                 ?psm.has-observables)
                 (set-role-value ?psm 'has-current-abstractors
                                 ?psm.has-abstractors)
                 (repeat
                  (in-environment
                   ((?ab . (achieve-generic-subtask
                            ?psm select-abstractor
                            'has-observables
                            ?psm.has-current-observables
                            'has-abstractors
                            ?psm.has-current-abstractors)))
                   (if (abstractor ?ab)
                     (do
                       (set-role-value
                        ?psm 'has-current-abstractors
                        (remove ?ab
                                (role-value ?psm 'has-current-abstractors)))
                       (in-environment
                        ((?obs . (the ?obs2 (has-current-observables ?psm ?obs2)))
                         (?ob . (achieve-generic-subtask ?psm one-step-abstraction
                                                         'has-abstractor ?ab
                                                         'has-observables ?obs)))
                        (if (and (observable ?ob)
                                 (not (member ?ob ?obs)))
                          (set-role-value ?psm 'has-current-observables
                                          (cons ?ob ?obs)))))
                     (return ?psm.has-current-observables)))))))))
```

The abstraction PSM makes use of two subtasks, `select-abstractor` and `one-step-abstraction`. The former selects an abstraction mechanisms from those available and the latter performs each individual abstraction inference. These tasks and the associated PSMs are fairly trivial and therefore we do not need to discuss them here. The full specification of these components is provided in Appendix 2.

6.4.2. *Solution Ranking*

The goal of task `rank-solutions` is to produce a list of solutions ordered according to the given match criterion - see next definition.

```
(def-class rank-solutions (goal-specification-task) ?task
  ((has-input-role :value has-candidate-solutions
                   :value has-observables
                   :value has-match-criterion)
   (has-output-role :value has-solutions)
   (has-candidate-solutions :type solution-space)
   (has-observables :type observables)
   (has-match-criterion :type match-criterion
                        :default-value default-match-criterion)
   (has-solutions :type solution-space)
   (has-goal-expression
    :documentation
    "The goal is to rank the classes according to the match criterion.
     The output should be a list of solutions, in which no solution follows
     one which is worse"
    :value (kappa (?task ?solutions)
                  (forall (?sol1 ?sol2)
                          (=> (and (member ?sol1 ?solutions)
                                   (member ?sol2 ?solutions)
                                   (precedes ?sol1 ?sol2 ?solutions))
                              (not (better-match-score ?sol2 ?sol1))))))))
```

The definition below shows the control regime of a PSM which can be used to solve the rank-solutions task.

```
(def-class rank-solutions-psm (composite-method) ?psm

  (....
   (has-body
      :value '(lambda (?psm)
                (do
                  (loop for ?candidate in ?psm.has-candidate-solutions
                      do
                      (set-role-value
                       ?psm has-sol-score-pairs
                       (cons (list-of ?candidate
                                      (achieve-generic-subtask
                                       ?psm basic-heuristic-match
                                       'has-observables ?psm.has-observables
                                       'has-candidate-solution ?candidate
                                       'has-match-criterion
                                           ?psm.has-match-criterion))
                             ?psm.has-sol-score-pairs)))
                  (map 'first
                       (sort ?psm.has-sol-score-pairs
                             '(kappa (?pair1 ?pair2)
                                     (better-match-score
                                      (second ?pair1)(second ?pair2)
                                      ?psm.has-match-criterion)))))))))
```

The above method uses subtask `basic-heuristic-match` to compute the score of each candidate solution and then sorts the resulting <solution, score> pairs according to the score comparison relation associated with the given match criterion. The specification of the `basic-`

`heuristic-match` task and a related PSM can be found in Appendix 2. The task itself is very straightforward: the interesting aspect of the matching process is not computational but ontological. In other words, our library provides a generic framework for heuristic matching, which is generic with respect to the various match criteria which can be defined. This approach facilitates the reuse and configuration of our library: it is much easier to configure an application by acquiring or selecting components than by modifying behaviour.

6.4.3. Refinement and refinement-PSM

The goal of this task is to take a solution at a given level of refinement, say n, and generate all its possible refinements, using the given set of refiners. If a solution at level n cannot be refined, the method returns the empty list.

```
(def-class refinement (goal-specification-task) ?task
  ((has-input-role :value has-candidate-solution
                   :value has-observables
                   :value has-refiners)
   (has-output-role :value has-refined-solutions)
   (has-candidate-solution :type solution)
   (has-observables :type observables)
   (has-refined-solutions :type solution-space)
   (has-refiners :type list)
   (has-goal-expression
    :value
    (kappa (?task ?solutions)
           (forall ?sol
                   (=> (member ?sol ?solutions)
                       (exists ?ref
                               (and (member ?ref ?task.has-refiners)
                                    (member ?sol
                                            (apply-refiner-operator
                                             ?ref
                                             ?task.has-candidate-solution)))))))))
```

The PSM given below provides a way to carry out solution refinement. It uses task `collect-applicable-refiners` to identify all the applicable refinements and task `apply-refiners` to generate the refined solutions. The definitions of these tasks can be found in Appendix 2.

```
(def-class vanilla-refinement-psm (decomposition-method) ?psm
  "This method applies all refiners applicable to the given input solution"
  ((has-body
    :value
    '(lambda (?psm)
       (in-environment
          ((?sol . ?psm.has-candidate-solution)
           (?refs . (achieve-generic-subtask
                     ?psm collect-applicable-refiners
                     has-solution ?sol
                     has-refiners ?psm.has-refiners)))
        (achieve-generic-subtask ?psm apply-refiners
                                 has-solution ?sol
                                 has-refiners ?refs)))))
```

6.5. Summing up

Figure 5 summarizes the main tasks and PSMs discussed in this section. As shown in the figure the two classification PSMs presented here share the same task-method structure. Essentially they provide alternative search methods on classification search spaces. This relative simplicity of the task-method structure is not accidental and it is unlikely to change even when extensions to the library are made. The lesson seems to be that once a sophisticated task ontology is provided, then different behaviours result mainly from the nature of the task, the chosen search methods and the selected match, solution admissibility and solution exclusion criteria.

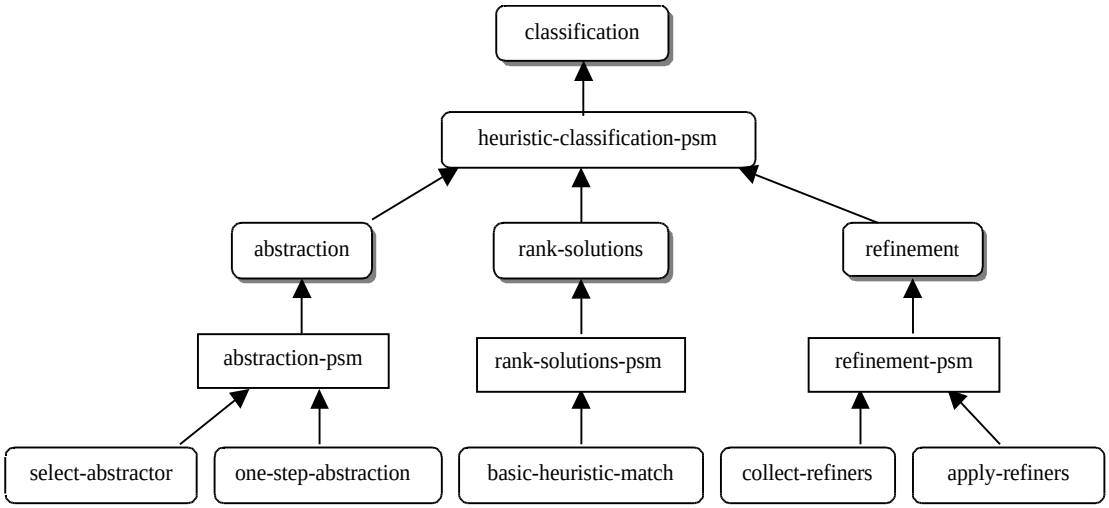

Figure 5. Main tasks and PSMs in the classification library

7. AN EXAMPLE: APPLE CLASSIFICATION

In this section we provide an example of the use of the library on a simple problem of apple classification. The goal of the application is to identify an unknown apple from its features, e.g., from her background colour, foreground colour, foreground pattern, size, taste, etc. The domain model we use has been provided by the University of Amsterdam and was developed prior to the development of the library. Hence it is totally independent of the task and method components of the library. The only changes we made to the original domain model were i) to translate if from a Prolog notation to OCML and to augment it with some new knowledge about types of apple, to allow us to illustrate the abstraction and refinement mechanisms. We should emphasize that these new types of apples added by us are complete fantasies.

7.1. Domain model

Our domain model consists of a hierarchy of apple types structured by means of class/subclass links. Figure 6 shows an illustrative subset of the domain model, which in total

contains 29 classes. Each class is described by a number of single-valued attributes, such as her colour, size, taste, etc. For instance, granny apples are represented as follows:

```
(def-class granny (apple)
  "Granny Smith"
  ((has_foreground :value no)
   (background :value green)
   (rusty :value no)))
```

It is important to emphasize that, in order to test the reusability of both the domain model and the library components, no changes were made to the given domain model, other than translating to OCML and augmenting it with some extra classes.

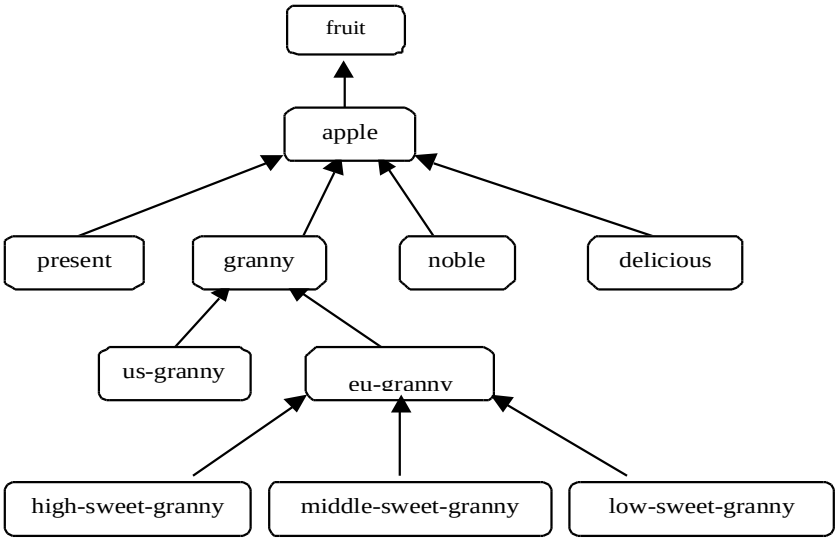

Figure 6. Illustrative subset of the domain model.

7.2. Application configuration

The application configuration process involves defining an application ontology, which comprises both the domain model and the task and PSMs specifications included in the library. These in turn are based on the associated ontologies. The resulting model inclusion structure is shown in figure 7. This shows that the apple classification application ontology also includes a generic task-domain mapping ontology, which configures the generic notions of solution feature for the class representation used in the apple domain model. Because such approach (representing solutions as classes and specifying features through class slots) is far from unusual, this generic mapping mechanism is provided as part of the classification library.

7.2.1. Task Configuration

In order to configure the generic specification of a classification task for a particular application the following types of knowledge must be instantiated:

- *Solution Classes*. In this scenario these will be the types of apples
- *Observables*. This provides a case input, i.e., an input to be provided each time we want to test the application for a particular case.
- *Match Criterion*. By default we use the four-dimensional criterion discussed in section 3.3.
- *Solution Admissibility Criterion*. The choice of this criterion tends to vary depending on whether we are interested in optimal or simply admissible solutions. In the latter case we need a strong criterion, such as complete coverage (i.e., all observables should be explained). In the former case a weaker criterion, such as positive coverage, may be sufficient.

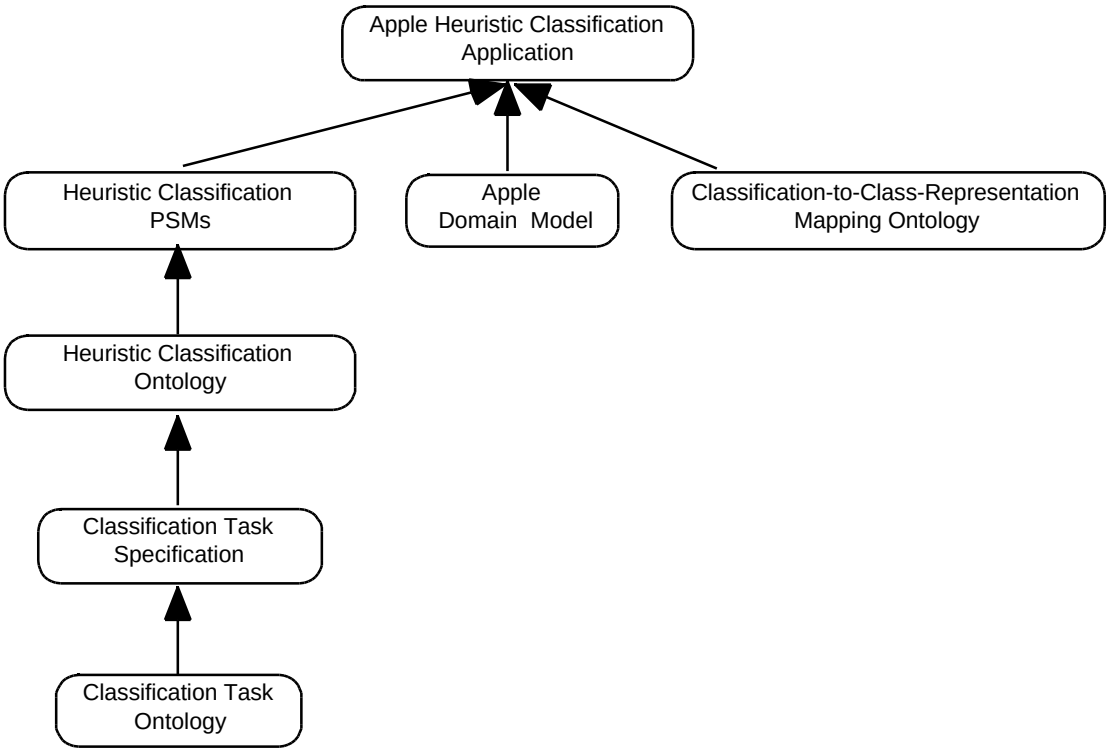

Figure 7. Model inclusion structure in the apple classification application.

OCML provides a *relation mapping* mechanism to support the process of defining bridges. Thus, we can link the task-level notion of `solution` to the domain-level notion of `apple` as follows:

```
(def-relation-mapping solution :up
  ((solution ?x)
   if
   (or (= ?x apple)
       (subclass-of ?x apple))))
```

The above definition defines an *upward* (or *read-only*) *mapping* from the domain level to the task level. Basically it says that something is a solution if it is either the class `apple` or one of its subclasses. OCML also provides mechanisms for defining *downward mappings* (i.e., to reflect down to the domain level knowledge inferred at the task level). More details on the OCML mapping mechanism can be found in (Motta, 1999).

Analogously we can use the relation mapping mechanism to link the generic notion of observable and the associated relations `has-observable-feature` and `has-observable-value` to the simple list-based notation we are going to use to input observables to our problem solver.

```
(def-relation-mapping observable :up
  ((observable ?x)
   if
   (== ?X ( ?f ?v))
   (or (and (slot-of ?f ?c)
            (or (= ?c apple)
                (subclass-of ?c apple)))
       (= ?f sugar))))

(def-relation-mapping has-observable-feature :up
  ((has-observable-feature ?ob ?f)
   if
   (== ?ob (?f ?v))))

(def-relation-mapping has-observable-value :up
  ((has-observable-value ?ob ?v)
   if
   (== ?ob ( ?f ?v))))
```

7.2.2. PSM Configuration

Our heuristic classification PSMs introduce three additional types of knowledge which have to be supplied by the domain.

- *Abstraction Mechanisms*. For illustration purposes we have augmented the domain model developed at the University of Amsterdam to include two attributes, `sugar-level` and `sweetness-level`, such that it is possible to perform qualitative abstraction from the former to the latter. The resulting abstraction mechanism is shown in the next section.
- *Refinement Mechanisms*. Given the class-based representation of the apple knowledge base, we can use the generic refiner `refinement-through-subclass-of-links`, shown in section 6.2.2.
- *Solution Exclusion Criterion.* We will use the default one provided by the library, which rules out inconsistent solutions.

7.2.3. Abstractors

The definition below shows the sample abstraction mechanism defined for the apple classification application. The definition abstracts the quantitative reading of the sugar level in an apple to a qualitative sweetness value, which can be one of {high, medium, low}. The abstraction mechanism can be applied if the input set of observables include the feature `sugar-level`.

```
(def-instance sugar-abstractor abstractor
  ((has-body '(lambda (?obs)
               (in-environment ((?v . (observables-feature-value
                                       ?obs 'sugar-level)))
                               (cond ((>= ?v 70) '(sweetness-level high))
                                     ((and (< ?v 70) (> ?v 40))
                                      '(sweetness-level medium))
                                     ((<= ?v 40) '(sweetness-level low))))))
   (applicability-condition (kappa (?obs)
                                   (member 'sugar-level
                                           (all-features-in-observables ?obs))))))
```

7.2.4. Results

We have tested the application with a variety of case inputs and criteria to verify the library and validate its applicability. The evaluation process has been satisfactory: the components appears to be robust and, as illustrated by the above discussion on the application configuration process, classification applications can be prototyped very quickly and very little additional application knowledge is required. Below, we show sample outputs from the apple classification application.

```
? (apple-single-sol-classification ' ((area china)
                                      (background green)
                                      (rusty  no)))
Result:  CHINESE-GRANNY


? (apple-optimal-classification
   '((background green) (area china) (sugar-level 30)))

Result:  (LOW-SWEET-GRANNY)
```

8. RELATED WORK

Classification problem solving has been investigated by many researchers (Clancey 1985, Clancey 1992, Stefik 1995, Wielinga et al. 1998, Althoff et. al.1994). The basic heuristic classification pattern was first identified by Clancey (1985), which also discussed several control strategies:

- *Data-directed search*. The program works forwards from data to abstraction, matching all solutions until all possible (or non-redundant) inferences have been made.
- *Solution-directed search.* The program works backwards from solutions, opportunistically collecting evidence to support them.
- *Opportunistic search*. The program combines data and solution-directed reasoning. Data abstraction rules tend to be applied as soon as data become available but the system is also able to reason in a solution-directed style. This approach is typical of *blackboard systems* (Hayes-Roth and Hayes-Roth, 1979).

The current version of our library only supports data-directed search, although we also plan to extend it with methods for solution-directed search in the future. From a more general point of view the main difference between our analysis and that of Clancey is that we use a much more sophisticated framework, which combines a generic epistemology of reusable components with the recent advances in the areas of ontologies and PSMs. As a result, our analysis is both more detailed and more principled. From an analytical viewpoint we provide a precise specification of a family of classification tasks, as well as the specification of the competence and domain assumptions associated with classification PSMs. From an engineering viewpoint our aim is to support application development by reuse. Hence, our components are also operational and can be rapidly configured for specific application domains. In particular, the separation between tasks, PSMs, domain models and applications provides us with a high degree of flexibility in organising and maintaining the library and in configuring it for specific applications.

Stefik (1995) provides a very good overview of classification problem solving in the context of his textbook on knowledge systems. Indeed one of our goals in developing the library was to provide effective support for the range of classification phenomena discussed by Stefik. A basic difference between our work here and Stefik's analysis is the one already highlighted when comparing our work with Clancey's: our aim is to provide an effective library resource, rather than simply to analyse classification problem solving. In addition, an important feature of our classification task specification is that we explicitly include the match and solution admissibility criteria as input roles of the task. Thus, the variations on the basic classification model discussed by Stefik can be obtained by providing different fillers for a classification task's input criteria. Our principled separation between task and PSM specification also allows us to separate clearly those aspects of Stefik's analysis associated with classification problems (e.g., match and admissibility criteria) from those associated with the problem solving process (e.g., the solution exclusion criterion). Another interesting consequence of including the criteria in the declarative specification of classification tasks and PSMs is that these criteria can then be reasoned about by the problem solver itself. For instance, it is relatively straightforward to define a flexible problem solver which starts with a complete coverage criterion and then relaxes it to a positive coverage one if no complete explanation is found.

Classification is also discussed in (Wielinga et al 1998). Their analysis of classification problem solving is carried out in the context of illustrating their competence theory for KBS development, the main goal of the paper. As a result their analysis only covers the basic of classification problem solving, although it is carried out in a formal way and makes interesting suggestions, such as the separation between contextual and non-contextual features.

The classification problem has also been considered by researchers in the case-based reasoning area (Althoff et. al. 1994). This work mainly focuses on similarity measures, which correspond to our match criterion. Indeed, in the aforementioned paper Althoff et. al. present a similarity measure function which is very similar to the default match criterion discussed in section 3.3.

9. CONCLUDING REMARKS

The work described in this document serves four main purposes:

- To carry out a thorough analysis of classification tasks and classification problem solving.
- To provide a useful engineering resource to support the development of classification problem solvers by reuse.
- To provide a concrete set of components on which to test future versions of the IBROW brokering system.
- To assess the suitability of OCML to support the IBROW approach and in particular to identify any possible discrepancies in the underlying meta-models.

We will discuss each of these objectives in turn.

9.1. Yet another analysis of classification tasks and classification problem solving

Classification problem solving has been thoroughly analysed in the literature, so it may seem a bit strange that new analyses are needed. Nevertheless it seems to us that our analysis may provide an additional useful contribution to the literature. It is the only analysis we know of which develops in detail a task and a PSM ontology for classification, thus highlighting and formalizing both the various types of knowledge needed for classification and the assumptions introduced by PSMs which perform a search in the space of candidate solutions.

9.2. A library of reusable components for classification

This is the most obvious contribution of this work to researchers and practitioners. Our classification library provides a practical resource which is freely accessible on the web through the WebOnto server and that can be used to develop applied classification problem solvers.

9.3. A testbed for the IBROW brokering system

The main aim of the IBROW project is to provide brokering support for knowledge component reuse over the World-Wide-Web. In order to test whether this goal is feasible we need to develop libraries organized according to the IBROW framework. The library described in this document fulfils this role. However, we should note that the current version of the library hardly provides any branching points (i.e., alternative PSMs) for any particular (sub-)task. We have already commented that this may be both a side-effect of the richness of our framework (we can achieve a great degree of differentiation in problem solving simply by providing different task specifications and criteria) and a feature of classification problem solving (i.e., it is a relatively simple class of problems). Nevertheless this relative simplicity of the library may be actually an advantage: in the first instance we can focus the work on developing intelligent support for acquiring a task specification from a user and operationalizing it in a scenario were only few problem solving alternatives are present.

9.4. IBROW and OCML

At a coarse-grained level of analysis there is very little difference between the IBROW framework for reuse (Fensel et al., 1999a; 1999b) and the TMDA framework which underpins the definition of the OCML language (Motta, 1999). This is of course not surprising given than one of us (Motta) has developed OCML and has also been one of the main contributors to the UPML specification. The main difference between the TMDA and UPML frameworks for reuse is that the former emphasizes the distinction between application-specific problem solving knowledge and mapping knowledge, while the latter appears to consider all application knowledge as part of the mapping knowledge required to define task-domain or PSM-domain bridges.

At a finer level of abstraction we need to check to what extent OCML supports the notions of refinements and bridges which are crucial to the UPML modelling philosophy.

9.4.1. Refinement mechanisms in OCML

Classes, as well as class-based entities, such as tasks and PSMs can be specialized (i.e., refined) using the standard class/subclass hierarchy provided by OCML. No refinement mechanism is available for functions, although of course sub-ontologies can override the definition of a function inherited from a super-ontology, thus providing a crude emulation of a refinement mechanism. The same situation applies to relations, although in cases in which a relation is defined through a set of rules, it can be refined simply by adding more rules.

9.4.2. Bridge mechanisms in OCML

Task and PSM specifications can be linked to domain definitions by means of the mapping mechanisms illustrated in section 7.2.1. OCML also provides *instance mapping* mechanisms which allow instances defined at the task level to be directly mapped to instances at the domain level. In our experience - see also (Motta, 1999) for examples from parametric design

applications - these mechanisms, which are normally part of a wider application ontology, are sufficient to bridge the gap between task and PSMs on one side and domain models on the other side.

Where OCML exhibit some weakness is in the support for mapping PSMs to tasks. The current meta-ontology characterizing these notions in OCML assumes that the input roles of a PSM include (and maybe extend) those of the task to which it is applied; no explicit role adaptation mechanism is provided, except of course that such adaptation can occur implicitly in the body of the PSM. As mentioned in section 6.3.1 a fairly simple adaptation mechanism is provided for output roles, through the combined use of the template slot `has-output-mapping` and the own slot `tackles-task-type`. This facility makes it possible to specify for which class of tasks a PSM has been defined and whether any further processing of the output of the PSM is required to conform to the requirement of the related generic task.

9.5. Final remarks

The library described in this document covers a range of classification phenomena and has been extensively tested and validated on the apple domain. Thus, it provides the necessary degree of robustness to be used for developing applications and as a test case for the IBROW broker. Future work will aim at extending the library, in particular by providing support for solution-directed approaches to classification. We also plan to try out the library on more complex problems: we are currently looking at a problem requiring the identification and selection of manufacturing technology and we also plan to investigate scientific classification problems.

The library is accessible online through the WebOnto web server. However, WebOnto only provides limited support for configuring and running PSMs. Future work will therefore also investigate ways to improve such support.

REFERENCES

Althoff, K.-D., Wess, S., Bergmann, R., Maurer, F., Manago, M., Auriol, E., Conruyt, N., Traphöner, R., Bräuer, M. and Dittrich, S. (1994): *Induction and Case Based Reasoning for Classification Tasks.* In: H. H. Bock, W. Lenski and M. M. Richter (eds.), Information Systems and Data Analysis, Prospects-Foundations-Applications, Proceedings of the 17th Annual Conference of the GfKl, University of Kaiserslautern, 1993, Springer Verlag, Berlin-Heidelberg, 1994, 3-16.

Benjamins, R. (1993). *Problem Solving Methods for Diagnosis*. PhD Thesis, Department of Social Science Informatics, University of Amsterdam.

Benjamins, V. R. and Fensel, D. (1998). Special Issue on Problem Solving Methods. *International Journal of Human Computer Studies*, 49(4), 1998.

Breuker, J. A. and Van de Velde, W. (1994). *CommonKADS Library for Expertise Modelling*. IOS Press, Amsterdam, The Netherlands.

Clancey W. J. (1985). Heuristic Classification. *Artificial Intelligence, 27*, pp. 289-350.

Clancey W.J. (1992) Model Construction Operators. *Artificial Intelligence* 53, pp. 1-115.

Domingue, J. (1998). Tadzebao and WebOnto: Discussing, Browsing, and Editing Ontologies on the Web. In B. Gaines and M. Musen (editors), *Proceedings of the 11th Knowledge Acquisition for Knowledge-Based Systems Workshop*, April 18th-23th, Banff, Canada.

Farquhar, A., Fikes, R., and Rice, J. (1996). The Ontolingua Server: A Tool for Collaborative Ontology Construction. In B. Gaines and M. Musen (Editors), *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*. Banff, Alberta, Canada.

Fensel, D., Benjamins, V. R., Motta, E. and Wielinga, B. J. (1999a). A Framework for knowledge system reuse. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99)*. Stockholm, Sweden, July 31 - August 5, 1999.

Fensel, D., Motta, E., Benjamins, V. R., Decker, S., Gaspari, M., Groenboom, R., Grosso, W., Musen, M., Plaza, E., Schreiber, G., Studer, R. and Wielinga, B. (1999b). The Unified Problem-solving Method Development Language UPML. IBROW3 Project Deliverable 1.1. Available online at http://www.aifb.uni-karlsruhe.de/WBS/dfe/publications99.html.

Gruber, T. R. (1995). Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies 43(5/6)*, pp. 907-928.

Guarino, N., Masolo, C. and Vetere, G. (1999). Ontoseek: Using large linguistic ontologies for accessing on-line yellow pages and product catalogs. *IEEE Intelligent Systems, 14(3)*, pp. 70–80.

Hayes-Roth, B. and Hayes-Roth, F. A Cognitive Model of Planning. Cognitive Science, 3, pp. 275-310.

Hori, M. and Yoshida, T. (1998). A Domain-oriented Library of Scheduling Methods: Design Principles and Real-life Applications. *International Journal of Human Computer Studies, 49(4)*, pp. 601-626.

McGuinness, D. (1998). Ontological issues for knowledge-enhanced search. In N. Guarino (editor), *Formal Ontologies in Information Systems*, IOS Press.

McGuinness, D. L. (1999). Ontology-enhanced Search for Primary Care Medical Literature. In *Proceedings of the International Medical Informatics Association (Working Group 6 - Medical Concept Representation and Natural Language Processing) Conference*, Phoenix, Arizona, December 16--19, 1999. Paper available online at URL http://ksl.stanford.edu/people/dlm/papers/imia-dlm-final.doc.

Motta E., Buckingham-Shum, S. and Domingue, J. (2000). Ontology-Driven Document Enrichment: Principles, Tools and Applications. *International Journal of Human-Computer Studies, 52(6)*, pp. 1071-1109.

Motta E. (1999). *Reusable Components for Knowledge Models: Principles and Case Studies in Parametric Design*. IOS Press.

Motta, E. and Zdrahal, Z. (1998). An approach to the organization of a library of problem solving methods which integrates the search paradigm with task and method ontologies. *International Journal of Human-Computer Studies 49(4)*, pp. 437-470. 1998.

Motta, E., Fensel, D., Gaspari, M. and Benjamins, R. (1999). Specifications of Knowledge Components for Reuse. *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering (SEKE '99)*, pp. 36-43. Kaiserslautern, Germany. Published by the Knowledge Systems Institute, 3240 Main Street, Skokie, IL, 60076 USA.

Mulholland, P., Zdrahal, Z. and Domingue, J. and Hatala, M. (2000). Integrating working and learning: a document enrichment approach. *Behaviour and Information Technology, 19 (3)*, 171-180.

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

Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Shadbolt, N., Van de Velde, W. and Wielinga, B (2000). *Knowledge Engineering and Management*. MIT Press. Cambridge, MA.

Speel, P-H. and Aben, M. (1997). Applying a library of problem solving methods on a real-life task. *International Journal of Human-Computer Studies 46*, pp. 627-652.

Stefik M. (1995). *Introduction to Knowledge Systems*. Morgan Kaufmann, San Francisco, CA.

van Heijst, G. (1995). *The Role of Ontologies in Knowledge Engineering*. PhD thesis, University of Amsterdam, May 1995.

Wielinga, B. J., Akkermans, J.K. and Schreiber, G. (1998). A Competence Theory Approach to Problem Solving Method Construction, *International Journal of Human-Computer Studies, 49*, pp. 315-338.