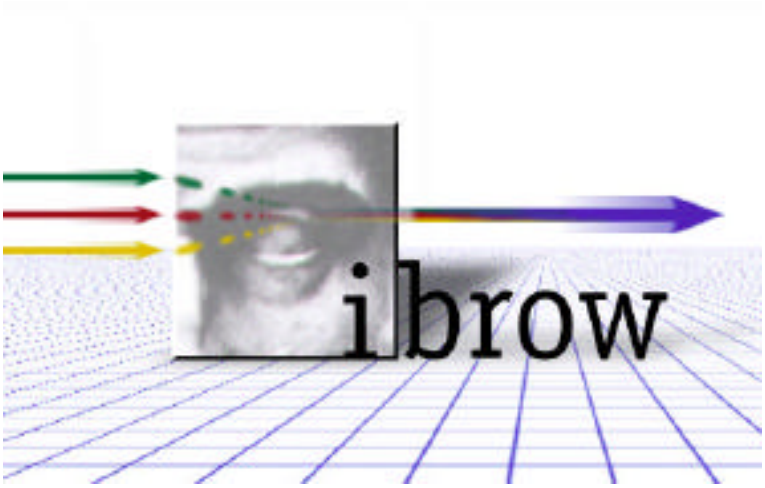

Appendix 1: Classification Task Specification

Enrico Motta[1], Wenjin Lu[1]

Abstract

In this appendix we show the complete specification of the classification task ontology and the definitions of a family of classification tasks.

Identifier:	**D1_Appx1**
Class:	**Deliverable**
Version:	**1.0**
Date:	**7 September, 2000**
Status:	**Draft**
Distribution:	**Project Members**
Responsible partner:	**The Open University**

```
;;;TASK CLASSIFICATION-TASK
(def-class classification-task (goal-specification-task) ?task
  "Classification is defined here as finding one or more admissible solutions
   out of a predefined solution space, which explain the features of a
   given set of observables, in accordance with a given match criterion and
   solution admissibility criterion.
   Because different variants of the goal can be formulated, the goal
   of the task is given here only as a default"
  ((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
    :documentation "A classification task is meaningful iff both observables and
                    solution space are provided"
    :value (kappa (?task)
                  (exists (?x ?y)
                          (and (member ?x (role-value ?task 'has-observables))
                               (member ?y (role-value ?task
                                                      'has-candidate-
solutions))))))
   (has-goal-expression
    :documentation "The goal is to find one or more admissible solutions"
    :default-value (kappa (?task ?sols)
                          (forall ?sol
                                  (=> (member ?sol (role-value ?task 'has-
solutions))
                                      (admissible-solution
                                       ?sol
                                       (apply-match-criterion
                                        (role-value ?task 'has-match-criterion)
                                        (role-value ?task 'has-observables)
                                        ?sol)
                                       (role-value
                                        ?task
                                        'has-solution-admissibility-
criterion)))))))

;;;SINGLE-SOLUTION-CLASSIFICATION-TASK
(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 (role-value ?task 'has-solutions))
                     2)))))
```

```
;;;TASK OPTIMAL-CLASSIFICATION-TASK
(def-class optimal-classification-task (classification-task) ?task
  "Another definition of a classification task which requires a solution to be
   optimal with respect to the given match criterion."
  ((has-goal-expression
    :documentation "The goal is to find the set of admissible solutions, say ?sols,
                    which best explain the observables with respect to the
                    given match criterion"
    :default-value (kappa (?task ?sols)
                          (forall ?sol
                                  (=> (member ?sol (role-value ?task 'has-
solutions))
                                      (and

                                       (admissible-solution
                                        ?sol
                                        (apply-match-criterion
                                         (role-value ?task 'has-match-criterion)
                                         (role-value ?task 'has-observables)
                                         ?sol)
                                        (role-value
                                         ?task
                                         'has-solution-admissibility-criterion))

                                       (best-match
                                        (role-value ?task 'has-observables)
                                        ?sol
                                        (role-value ?task 'has-candidate-solutions)
                                        (role-value ?task 'has-match-
criterion)))))))))

;;;CLASS OBSERVABLES
(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))))))

;;;CLASS OBSERVABLE
(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)))
```

```
;;;RELATION LEGAL-FEATURE-VALUE
(def-relation legal-feature-value (?f ?v)
  "True if ?v is a legal value for a feature ?f")

;;;FUNCTION ALL-FEATURES-IN-OBSERVABLES
(def-function all-features-in-observables (?obs)
  "This function extracts the features from a set of observables"
  :body (setofall ?f (and (member ?ob ?obs)
                          (has-observable-feature ?ob ?f))))

;;;FUNCTION OBSERVABLES-FEATURE-VALUE
(def-function 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"
  :body (if (and (member ?ob ?obs)
                 (has-observable-feature ?ob ?f))
          (the ?v (has-observable-value ?ob ?v))))

;;;CLASS FEATURE
(def-class feature ())

;;;CLASS CONTEXTUAL-FEATURE
(def-class contextual-feature (feature)
  "Wielinga et al (IJHCS, 49, 1998) point out that feature classification is
   context-dependent.
   For instance, a red apple in a supermarket in summer may be a GOLDEN_APPLE,
   while a red apple on a tree in summer may not feasibly be a GOLDEN_APPLE, given
   that these are only found on trees in winter (we know nothing about apples, the
   example is completely made up.....).
   In this ontology we model context knowledge has simply another class of
   features. Certain features are contextual, other are not.  It will be up to the
   classification problem solving process, which is not part of this task ontology,
   to decide how to handle contextual features")

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

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

;;;CLASS FEATURE-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))))
```

```
;;;FUNCTION ALL-FEATURES-IN-SOLUTION
(def-function all-features-in-solution (?sol)
  "This returns all the features in a solution"
  :body (setofall ?f  (and (element-of ?fd ?sol)
                           (has-feature-name ?fd ?f))))

;;;FUNCTION SOLUTION-FEATURE-SPEC
(def-function solution-feature-spec (?sol ?f)
  "This returns the spec associated with a particular feature in a solution"
  :body (the ?spec (and (member ?fd ?sol)
                        (has-feature-name ?fd ?f)
                        (has-feature-value-spec ?fd ?spec))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;

;;;BEST-MATCH
(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 metrics - 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))))))

;;;BETTER-MATCH-THAN
(def-relation better-match-than (?sol1 ?sol2 ?obs ?criterion)
  "?sol1 is a better solution than ?sol2, according to ?obs and ?criterion"
  :constraint (and (observables ?obs)
                   (solution ?sol1)
                   (solution ?sol2)
                   (match-criterion ?criterion))
  :iff-def (and (sol-has-match-score ?sol1 ?obs ?score1 ?criterion)
                (sol-has-match-score ?sol2 ?obs ?score2 ?criterion)
                (better-match-score ?score1 ?score2 ?criterion)))

;;;SOL-HAS-MATCH-SCORE
(def-relation sol-has-match-score (?sol ?obs ?score ?criterion)
  "?score expresses a measure - with respect to some metrics -
   of the extent to which ?class covers ?obs with respect to"
  :constraint (and (observables ?obs)
                   (solution ?sol)
                   (match-score ?score)
                   (match-criterion ?criterion))
  :iff-def (= ?score (apply-match-criterion ?criterion ?obs ?sol)))

;;;MATCH-SCORE --- All measures for assessing the match between a class
;;;and an instance should be defined as subclasses of this class
(def-class match-score ())
```

```
;;;MATCH-SCORE-TYPE
(def-class match-score-type () ?x
  :iff-def (subclass-of ?X match-score))

;;;DEFAULT-MATCH-SCORE
(def-class default-match-score (match-score) ?s
  "This class quantifies the match between a class C and an instance I
   in terms of a 4-place vector:
   <inconsistent-features, explained-features, unexplained-features, missing-
features>
    Feature is INCONSISTENT - f1 is in I and in C but their values are inconsistent
    Feature is EXPLAINED - f1 is in I and in C and their values are consistent
    Feature is UNEXPLAINED - f1 is in I but not in C
    Feature is MISSING - f1 is in C but not in I"

  :iff-def (and (list ?s)
                (= (length ?s)
                   4)))

;;;BETTER-MATCH-SCORE
(def-relation better-match-score (?d1 ?d2 ?criterion)
  "true if ?d1 is a better score than ?d2 according to ?criterion"
  :constraint (and (match-score ?d1)
                   (match-score ?d2)
                   (match-criterion ?criterion))
  :axiom-def (defines-partial-order better-match-score)
  :iff-def (holds (the ?rel (has-match-score-comparison-relation ?criterion ?rel))
                  ?d1
                  ?d2))

;;;CLASS MATCH-CRITERION
(def-class match-criterion ()
  "A match criterion has a scoring mechanism and a match-score-comparison relation.
   The former is needed to assess the degree of match between a solution and a set
   of observables.  The latter to compare two scores"
  ((has-scoring-mechanism :type match-score-mechanism)
   (has-match-score-comparison-relation :type match-score-comparison-relation)))

;;;CLASS MATCH-SCORE-MECHANISM
(def-class match-score-mechanism ()
  "A match-score mechanism has two levels.  A level at which each <feature value>
   pair is scored, and a level at which scores are combined"
  ((has-macro-scoring-mechanism  :type macro-score-mechanism)
   (has-feature-scoring-mechanism  :type feature-score-mechanism)))

;;;CLASS MATCH-SCORE-COMPARISON-RELATION
(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)))

;;;RELATION DEFAULT-MATCH-SCORE-COMPARISON-RELATION
(def-relation default-match-score-comparison-relation (?d1 ?d2)
  :iff-def (= (default-the-better-match-score ?d1 ?d2) ?d1))
```

```
;;;FUNCTION THE-BETTER-MATCH-SCORE
(def-function the-better-match-score (?d1 ?d2 ?criterion)
  :body (in-environment
         ((?rel . (the ?rel (has-match-score-comparison-relation ?criterion
?rel))))
         (cond ((holds ?rel ?d1 ?d2)
                ?d1)
               ((holds ?rel ?d2 ?d1)
                ?d2)
               ((true)
                :equal))))

;;;FUNCTION APPLY-MATCH-CRITERION
(def-function apply-match-criterion (?criterion ?observables ?solution)
  :body (in-environment
         ((?scoring-mechanism . (the-slot-value ?criterion has-scoring-mechanism))
          (?macro-mech . (the-slot-value ?scoring-mechanism
                                         has-macro-scoring-mechanism))
          (?micro-mech . (the-slot-value ?scoring-mechanism
                                         has-feature-scoring-mechanism)))
         (call ?macro-mech ?observables ?solution ?micro-mech)))

;;;INSTANCE  DEFAULT-MATCH-CRITERION
(def-instance default-match-criterion match-criterion
  ((has-scoring-mechanism  default-scoring-mechanism)
   (has-match-score-comparison-relation default-match-score-comparison-relation)))

;;;INSTANCE DEFAULT-SCORING-MECHANISM
(def-instance default-scoring-mechanism match-score-mechanism
  ((has-macro-scoring-mechanism default-macro-score-mechanism)
   (has-feature-scoring-mechanism  default-feature-score-mechanism)))


;;;CLASS MACRO-SCORE-MECHANISM
(def-class macro-score-mechanism (function) ?f
  "This is a function which takes as input a set of observables, a solution and
    a feature-level scoring mechanism and returns a match score")


;;;FUNCTION DEFAULT-MACRO-SCORE-MECHANISM
(def-function default-macro-score-mechanism (?observables ?solution
                                                          ?feature-score-mech)
  :lisp-fun #'(lambda (observables solution feature-score-mech)
                (default-macro-score-mechanism observables solution
                  feature-score-mech)))
```

```
(defun default-macro-score-mechanism (observables solution feature-score-mech)
  (let ((obs-features (ocml-eval-gen `(all-features-in-observables ',observables)))
        (sol-features (ocml-eval-gen `(all-features-in-solution ',solution)))
        (score-vector (list nil nil nil nil)))
   ;;initialize unexplained features (in i but not in c)
    (setf (third score-vector)
          (set-difference obs-features
                          sol-features))
    (loop for feature in obs-features
          for result = (ocml-eval-gen
                        `(call ',feature-score-mech ',observables ',feature
                               ',solution ',sol-features))
          do
          (case result
            (:inconsistent (if (not (member feature (first score-vector)))
                               (setf (first score-vector)
                                     (cons feature (first score-vector)))))
            (:explained (setf (second score-vector)
                              (cons feature (second score-vector))))
            (:unexplained  (if (not (member feature (third score-vector)))
                               (setf (third score-vector)
                                     (cons feature (third score-vector))))))
          finally
          (progn
            (setf (fourth score-vector)
                (set-difference sol-features obs-features))
            (return score-vector)))))

;;;CLASS FEATURE-SCORE-MECHANISM
(def-class feature-score-mechanism (function) ?f
  "This is a function which takes as input a set of observables, a feature,
   a solution and the list of features associated with the solution and returns
   a feature-level score")

;;;FUNCTION DEFAULT-FEATURE-SCORE-MECHANISM
(def-function default-feature-score-mechanism (?observables ?feature ?solution
                                                            ?solution-features)
  :body (in-environment ((?fv . (observables-feature-value ?observables ?feature)))
                        (if (member ?feature ?solution-features)
                          (in-environment ((?f-spec . (solution-feature-spec
                                                       ?solution ?feature)))
                                          (if (= ?f-spec :nothing)
                                            :unexplained
                                            (if (holds ?f-spec ?fv)
                                              :explained
                                              :inconsistent)))
                          :unexplained)))

;;;DEFAULT-THE-BETTER-MATCH-SCORE
(def-function default-the-better-match-score (?s1 ?s2)
  "returns the better match score between ?s1 and ?s2 or :equal
   if they are the same score.
   The better score is the one which minimises inconsistency,
   maximises explanation power and minimises missing features.
  These criteria are applied in order"
  :constraint (and (default-match-score ?s1)
                   (default-match-score ?s2))
  :lisp-fun #'(lambda (x y) (default-the-better-match-score x y)))
```

```
;;;DEFAULT-THE-BETTER-MATCH-SCORE
(defun default-the-better-match-score (s1 s2)
  "Returns the better match score between ?s1 and ?s2 or :equal
   if they are the same score.
   The better score is the one which minimises inconsistency,
   maximises explanation power and minimises missing features.
   These criteria are applied in order"
    (let*
         ((inconsistent1 (first s1))
          (explained1 (second s1))
          (unexplained1 (third s1))
          (missing1 (fourth s1))
          (inconsistent2 (first s2))
          (explained2 (second s2))
          (unexplained2 (third s2))
          (missing2 (fourth s2)))

         (cond ((< (length inconsistent1)
                   (length inconsistent2))
                s1)
               ((< (length inconsistent2)
                   (length inconsistent1))
                s2)
               ((< (length explained1)
                   (length explained2))
                s2)
                ((< (length explained2)
                   (length explained1))
                s1)
                ((< (length unexplained1)
                    (length unexplained2))
                s1)
                ((< (length unexplained2)
                    (length unexplained1))
                s2)
                ((< (length missing1)
                    (length missing2))
                s1)
                ((< (length missing2)
                    (length missing1))
                s2)
                (t
                 :equal))))
```

```
;;; CLASS SOLUTION-ADMISSIBILITY-CRITERION
(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)))

;;;INSTANCE  DEFAULT-SOLUTION-ADMISSIBILITY-CRITERION
(def-instance default-solution-admissibility-criterion solution-admissibility-
criterion
 ((applies-to-match-score-type default-match-score)
  (has-solution-admissibility-relation default-solution-admissibility-relation)))

;;;RELATION DEFAULT-SOLUTION-ADMISSIBILITY-RELATION
(def-relation default-solution-admissibility-relation (?score)
  "A solution should not have any inconsistent features and should explain at least
   one feature"
  :constraint (default-match-score ?score)
  :iff-def (and (= (length (first ?score)) 0)
                (> (length (second ?score)) 0)))

;;;INSTANCE COMPLETE-COVERAGE-ADMISSIBILITY-CRITERION
(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)))

;;;RELATION COMPLETE-COVERAGE-ADMISSIBILITY-RELATION
(def-relation complete-coverage-admissibility-relation (?score)
  "a solution should be consistent and explain all features"
  :constraint (default-match-score ?score)
  :iff-def (and (= (length (first ?score)) 0) (= (length (third ?score)) 0)))

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

```
;;;AXIOM ADMISSIBILITY-IS-MONOTONIC
(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))))

;;;RELATION COMPOSABLE-SOLUTIONS
(def-relation composable-solutions (?sol1 ?sol2)
  "Two solutions are composable if they are not mutually inconsistent with respect
   to some feature.
   This definition is not operational"
  :no-op (:iff-def (not (exists (?f ?v)
                        (and (member ?f (all-features-in-solution ?sol1))
                             (member ?f (all-features-in-solution ?sol2))
                             (holds (solution-feature-spec ?sol1 ?f) ?v)
                             (not (holds (solution-feature-spec ?sol2 ?f) ?v)))))))
```