An Object-Oriented Approach to Classification

Amedeo Napoli

LMIAS, Institute Le Bel, Université Louis Pasteur, 67000 Strasbourg, France

Roland Ducournau

SEMA Group, 92 Montrouge, France

Claude Laurenco

CRIM, 34100 Montpellier, France

ABSTRACT

We present in this paper two possible semantics for the subsumption relation. On the one hand, the subsumption relation is equivalent to the inheritance relation between concepts. On the other hand, we define a subsumption relation that links concepts with other concepts, or with individual instances, based on matching components between the linked entities. According to the first of this dual view, we have developed a subsumption operation used for checking the consistency of inheritance graphs. According to the second, a different subsumption operation is used for enhancing the information retrieval and the problem-solving capabilities of a knowledge-based system. Next we characterize object oriented languages in general, followed by the particular language we use, which integrates frame-based and class-based features. We then present two classification-based algorithms, corresponding to each use of the subsumption relation, and discuss the way these are implemented using an object-oriented approach. We end the paper by describing our particular problem-solving application in the domain of organic chemistry.

THE SUBSUMPTION RELATION REVISITED

One of the practical applications of classification is maintenance, in particular updating of knowledge bases where real world concepts are described by objects organized in an inheritance hierarchy. Inheritance refers to sharing of properties by all objects below the ones where the properties are stored. Implementation of inheritance using class or frame data structures is discussed in the next section of this paper.

Updating a knowledge base with a new concept requires two basic tasks. First, the new concept must be described, and second, it must be placed in the correct location in the hierarchy. For the second task, the classifier, whether human or automated, must retrieve, in relation to the new concept, its immediate ancestors in the existing hierarchy. In addition, if these ancestors had specializations in the original hierarchy, these would be checked to see if they should be immediate specializations of the new concept instead. This classification process is achieved by a subsumption operation which compares descriptions of concepts to check if the concepts are related to one another according to this subsumption relation. Details of this algorithm are provided in the section Classification-Based Processes.

Let us recall the definition of the subsumption relation as stated in term subsumption languages (languages of the KL-ONE family) [Schmolze 83] [Brachman 85]: a concept *A* subsumes a concept *B* only if the set denoted by *A* necessarily includes the set denoted by *B*. The view presented in [Finin 86] is similar: *A* subsumes *B* if whatever is represented by the description *B* is also represented by the more general description *A*. For example, the concept *Food* subsumes the concept *Fruit*, the set of all fruits is included in the set of foods.

Generally, this subsumption relation is implemented as an inheritance relation. In both expressions above, all characteristics of A are inherited by B: a fruit is a food and it inherits food's properties such as the fact that a food does not kill anyone who eats it.

For problem-solving applications, where specific components of descriptions (i.e., categories of components) of concepts are designated *a priori* as having special significance for solving a problem, it can be useful to define subsumption as follows: a concept *A* subsumes a concept *B* if *A* contains in its description an important component of the description of *B*. For example, the concept *Flour-containing-food* contains in its description an important component, namely the ingredient *Flour*, which is necessarily contained in the description of *Bread*.

A practical application of this view is to enhance pattern-directed problem-solving capabilities of a knowledge-based system. A subsumption operation, differing from the one for the first semantics, is used which determines that an individual *B* is subsumed by a concept *A* if one of the components of *B* matches *A*. In this case, if *B* represents a goal to be reached, problem-solving operations associated with *A* can be applied to solve *B*. Details of this algorithm are provided in the last section of this paper, describing our actual chemistry problem-solving application. For example, suppose that we want to make *bread-314*, then the cooking method associated with *Bread* can be applied to *bread-314* in order to make it. This example shows our extension to the above definition to include not only linking two concepts, but also linking concepts with individual instances.

This view of subsumption, unlike the first subsumption relation, does not have any implications for inheritance. That is, concept *B* does not necessarily inherit properties from concept *A*. In fact, *A* might be an actual component of *B*. For example, in some problem-solving application, it might be said that "*Flour* subsumes *Bread*" if this proved to be useful for solving a problem. This statement would be analogous to "*C=O* (a chemical bond between carbon and oxygen) subsumes *Formaldehyde* (a molecule containing this structure)" in our actual chemistry problem-solving application, described in the final section of this paper.

INHERITANCE SYSTEMS

An inheritance system allows one to represented knowledge as a hierarchy of objects where each object can be seen as a collection of properties describing a concept [Touretzky 86]. An object encapsulates data and procedures acting on these data. Objects are partially ordered according to an inheritance relation: each object inherits the properties of one or more ancestors, i.e., single or multiple inheritance, the ancestors' properties being virtually "added" to those of the object. The inheritance relationship is transitive: each ancestor itself inherits the properties of one or more ancestors, and so on. The inheritance hierarchy forms a graph which has an upper bound corresponding to the most general object called the root of the inheritance graph.

In addition to describing concepts, objects may be descriptions of individual instances. For example, the concept *Person* can have the specializations *Parent* and Grandparent, and the instances *Maria* or *Person-4* which represent individuals. *Person* can be considered a template

ISSN: 2324-9773

from which its instances are made. Instances are always leaves in the hierarchy, whereas all internal nodes are concepts.

Inheritance systems are implemented using specialized computer languages which have characteristic data structure. Two types of language are currently employed, *class-based* and *frame-based.*

A class can be thought of as a template describing the structure and behavior of a set of instances. The structure of an instance is given by a set of instance variables while its behavior is represented by methods activated by means of message passing. A class may have one or several superclasses from which it inherits its structures and behaviours. There are two sorts of properties that may be inherited, structural properties and behavioral properties. Structure inheritance is static, and it fixes an instance structure at its class creation. Inheritance of behavior is dynamic, i.e., methods are attached to a class and are selected at run-time. Thus, an instance can only be created when its class has been defined. Moreover, an instance is modeled after its class and cannot dynamically change its class. These features make it difficult to perform classification-based reasoning in a simple way, in contrast to frame-based languages.

Frame-based languages are based on prototype theory which relies on the following assumption: a family of real objects can be represented by one typical well-known object, called *prototype*, which determines the majority of properties shared by members of the family [Cohen 84]. Each object is a specialization of one or more prototypes and can itself be a prototype. All objects are linked in an inheritance hierarchy, and they dynamically inherit their ancestors' properties. Frame-based languages do not normally have a built-in distinction between frames describing concepts and frames describing instances. An instance corresponds to a leaf in the inheritance graph, and is usually the description of an individual.

Real-world knowledge is rather difficult to represent within a single representation formalism because each has characteristic advantages and disadvantages. Representation languages based on objects have therefore evolved towards hybrid languages which integrate frames, classes, and rules formalisms. In the following dection we describe the hybrid language YAFOOL that we use in our application.

YAFOOL: AN OBJECT-ORIENTED LANGUAGE FOR KNOWLEDGE REPRESENTATION

YAFOOL is an object-oriented language of the knowledge engineering system Y3, which includes YAFEN, a graphic programming environment, and YAFLOG, a Prolog-like inerence engine. It is a hybrid object-oriented language: a frame-based language which possesses class-based language features [Ducournau 89]. An object is a frame composed of a collection of slots denoting the properties of a concept. There are two types of slot, attribute slots and method slots, which, respectively, describe the characteristics and the behavior of the concept. Attributes are annotated by declarative or procedural facets. As is usual in frame-based languages, inheritance of any property, attribute, or method is dynamic [Ducournau 87]. Inheritance paths are formed by the *is-a* slot which links frames in an inheritance hierarchy.

Let us consider for example the frame definitions of the concepts *Food* and *Flour-containing-food*:

ISSN: 2324-9773

```
(defmodel Food
   (is-a ($value Ideal-object))
   (weight ($a Float))
   (price ($a Float))
   (creation ($method food-create)))

(defmodel Flour-containing-food
   (is-a ($value Food))
   (shape ($a Symbol)
            ($domain long round with-a-hole without-hole))
   (cooking-time ($a Float))
   (ingredient ($a Ingredient)
                ($domain Flour Water Salt)))
```

The frame *Food* is a subframe of the most general, system-defined frame called *Ideal-object*. Each frame (except this most general frame) is a subframe of another frame. This second frame is an actual value introduced by the *$value* facet in the *is-a* slot of the subframe. In the above example, the frame *Food* is a subframe of the most general frame *Ideal-object*. Similarly, the frame *Flour-containing-food* is a subframe of *Food*. This subframe linkage implements inheritance; that is, *Flour-containing-food* inherits the properties of *Food*, and in turn, *Food* inherits the properties of *Ideal-object*.

Facet semantics are nearly classical [Masini 90]. The declarative facet $value is used to introduce the actual value of the attribute, which can be used both as a default or a fixed value. The frame linked to the current frame by is-a is introduced by this $value facet. Other declarative facets are used to describe the value of the attribute. These are divided into typing facets, $a and $list-of, and restriction facets, $domain and $interval. For example, the attribute *ingredient* in the frame *Flour-containing-food* has the type *Ingredient*, which is a frame presumed to be defined elsewhere. The $domain facet attached to this attribute declares the three basic ingredients which can be found in a flour-containing food.

Four procedural facets introduce *reflexes* (also known as *demons*) which are triggered when a slot is accessed, i.e., reading (when a value is read from a slot) or writing (when a value is written into a slot). The $if-needed facet holds a reflex to be run whenever a value is needed but none is present or cannot be inherited. The $required facet introduces a boolean reflex which checks the validity of a value to be written into the slot. This reflex is particularly employed to implement constraints that cannot be given by typing facets because they hold on several slots. The $if-added facet introduces a reflex to be run whenever a value is given or added to the slot. In a similar way, the $if-removed facet holds a reflex to be run whenever a value is removed from the slot.

Methods are introduced by the special facet $method. The slot name corresponds to the *selector* of the method, which is activated by message passing. For example, the method associated with the selector *creation* in the frame *Food* is used to create instances (e.g., *food-1*).

Let us consider for example the frames *Bread* and *Pasta*:

ISSN: 2324-9773

```
:ad-cooking-time)))



ıta-cooking-time)))
```

The value of the attribute *cooking-time* is computed by an if-needed reflex (a reflex is named according to the facet holding it) which is different for the frame *Pasta* and *Bread* (we suppose that this computation depends on the shape and the weight of the instances manipulated). These two frames also have a method the selector of which is *cook.* The method is implemented by two different functions, the cooking mode being different for Bread and Pasta.

Programming consists of slot accesses and message passing using special primitives called *applicators.* According to the type of slot accessed, whether attribute or method, the *::* applicator respectively performs a read access or a message passing. Standard reading mode is done according to Z inheritance [Winston 84]: an attribute value is first searched in the $value facet, and if there is none, in the $if-needed facet. A search is done level by level until a value is delivered or the root of the inheritance graph is reached. A value computed by an if-needed reflex is written into the attribute $value facet. For example, consider the following expressions (*?* is the user prompt; = precedes the system reply):

```
? (:: creation Bread '(shape long) '(weight 1))
= bread-1
? (:: shape bread-1)
= long
```

The first expression is the sending of the message, naming the selector *creation*, to the frame *Bread,* with initialization of the attributes *shape* and *weight.* This selector does not appear explicitly in the frame *Bread,* but is inherited by this frame from the frame *Food* via the is-a link from *Bread* to *Food.* The result of applying this method, i.e, running the *food-create* function, is an instance of the frame Bread named *bread-1* (this name is assigned automatically by the system). The second expression is an access to the value of the slot *shape* in the newly-created instance *bread-1.* An example of solving a problem by sending a message specifying the selector *cook* is presented in the next section of this paper.

Applicators *:=* and *:+:* are used to write a slot value. Writing is a three-step operation: constraints satisfaction checking, filling the slot value, and if-added reflex activation. All constraints, inherited or not, specified either by typing facets or required reflexes, must be satisfied. Constraints are checked in top-down order, and no writing is done if any one of the constraints is not satisfied. After the filling step, the if-added reflexes, inherited reflexes included, are run in top-down order (the same order as constraints checking). The applicator *:-* is used to remove a slot value. This operation consists of removing the value and then triggering the if-removed reflexes. All if-removed reflexes, inherited or not, are triggered in bottom-up order. The semantics of reflexes are based on the assumption that reflexes have the same generality as

ISSN: 2324-9773

the frames they belong to. Thus, reflexes must be run in an up-to-date environment: operations linked to specific information must be executed in an environment where general information is up-to-date.

YAFOOL is a reflective language: each entity is represented by a frame which contains information about itself [Maes 87]. Thus slots and facets are themselves objects which are organized in a multiple inheritance graph whose root is called *Ideal-dual*, a subframe of *Ideal-object*. Reflection permits associating meta-knowledge with each frame. For example, each slot contains an attribute which lists the frames it is part of, and each frame contains an attribute whose value is the ranked list of its important slots. This reflective knowledge is used for classification-based processes, as described in the next section.

CLASSIFICATION-BASED PROCESSES

Our approach to classification is inspired by the works done work done on KL-ONE [Lipkis 82]. However, our particular development is based on the two semantics of the subsumption relation as presented at the beginning of this paper: the first is used for checking consistency of inheritance graphs, while the second is used in problem-solving.

To check for subsumption between two concepts, i.e., to determine that the subsumption relation holds, we have defined a subsumption operation. This operation is distributed and is performed by means of message passing: a subsumption operation method is attached to the root of the inheritance graph and can be specialized for any particular frame. The most general method checks if a frame *A* is a subsumer of the frame *B* (the subsumee): all valued slots of B, inherited slots included, for consistency with corresponding slots of *A*. Specifically, slot values in the subsumee frame must be consistent with slot value requirements, e.g., type restrictions and cardinality, declared in the subsumer frame. For example, the system can check that the frames *Bread* and *Pasta* are well-defined. An if-needed reflex in *Pasta* is run to compute the value of the attribute *cooking-time*; there are no other attribute redefinitions for this particular frame. In the frame *Bread*, the domain associated with the attribute *shape* is redefined and checked to be sure it is included in the *shape* domain declared in the frame above it in the hierarchy, namely *Flour-containing-food*.

The fact that the subsumption operation is distributed is an advantage because it permits redefining this operation according to the particular characteristics of each frame. A frame *A* subsumes a frame *B* if each component of *A* subsumes each component of *B*. As said above, all entities are represented by objects, and the subsumption operation can be particularized for any object. In particular, an attribute *a1* in *A* subsumes the corresponding attribute *a2* in *B* if the values of the facets attached to *a1* subsume the values of the facets attached to *a2*. The subsumption operation is defined on the facets as follows: the type of *a1* subsumes the type of *a2*: the domain or interval associated with *a2* must be included in the corresponding domain or interval defined in *a1*; a value given by a $value facet being considered as a default value must have a type which is in accordance with the attribute's typing facet.

Suppose we have to add a new concept, named *C*, in an the inheritance graph (we can also consider *C* an existing concept to be reclassified). We first consider the list of frames that share

ISSN: 2324-9773

at least one slot with *C*. This list is established using reflective information associated with frames that describe the slots of *C*. The list contains the candidate subsumer frames. It is sorted in order to put the most general frames, i.e., those frames sharing the fewest slots with *C*, at the front of the list. Frames which do not share any slot with the concept are ignored. Using the operation described in the preceding paragraph, each frame of the candidate list is then tested to check whether it subsumes *C*. If the current frame does not subsume the concept, none of its specializations could either because of transitivity of the inheritance relation: frames in the subgraph whose root is the current frame will be ignored and thus are removed from the candidate subsumers list. If the current frame subsumes the concept, it is written to a list containing the potential subsumers. This process continues until the candidate subsumers list is empty. The list of potential subsumers is then sorted in order to keep only the most specific frames, which become the most specific subsumers. The most general subsumees are found amongst the specializations of the most specific subsumers in the same way, testing if C subsumes these specializations.

Suppose we want to classify the following general description

```
(defmodel X
   (is-a ($value Ideal-object))
   (shape ($value long with-a-hole))
   (ingredient ($value Flour Water)))
```

The list of candidate subsumers will be defined by the intersection of the frames list defining the attributes *shape* and *ingredient* (system attributes such as *is-a* are not considered). This list is *(Flour-containing-food Bread Pasta)*. The frame *Flour-containing-food* subsumes *X* because attribute values of *X* are consistent with value requirements declared in *Flour-containing-food*, i.e., the value for *shape* is included in the domain associated with *shape* in *Flour-containing-food*; the value for *ingredient* is included in the domain associated with *ingredient* in *Food*, which is inherited by *Flour-containing-food*. Analogously, the frame *Pasta* subsumes *X*. However, the frame *Bread* does not subsume *X* because the the attribute value for *shape* does not correspond to any value declared in the domain associated with *shape* in *Bread*. The frame *Pasta* being a subframe of *Flour-containing-food*, the former remains the only frame in the candidate list, and it becomes the subsumer of *X*.

The second classification-based application is an indexing application. In this indexing process, we are concerned with finding frames in the knowledge base that describe a particular instance frame presented to the system. These knowledge base frames presumably have associated methods which can then be used in a problem-solving strategy with respect to the given instance. The subsumption operation for this application differs from the one above in the following respect: only *a priori* designated important slots are considered in the first step of matching the given instance, i.e., the instance need not be fully subsumed by matching knowledge base frames.

Suppose that the goal of a problem is to build the product described by the previously presented frame *X*. We also know that building a product is performed via the sending of a message specifying the selector *cook*. We found that the frame *Pasta* subsumes the frame *X*. Thus the problem can be solved by applying to the frame *X* the cook method associated with

ISSN: 2324-9773

Pasta. The result of this application is the solution to the current problem. This example only briefly sketches the problem-solving scheme that we use to build a synthetic molecule. We present this scheme in the following section.

CLASSIFICATION FOR CHEMICAL PROBLEM-SOLVING

Our system is a computer-assisted synthesis system aimed at building up organic molecules, called target molecules, from readily available starting materials [Laurenco 90]. Once a target molecule has been chosen, the system searches for a synthetic plan which is defined by one or more synthetic paths, each of them constituted by a sequence of reactions leading from starting materials to the target molecule. A synthetic path is generally made up using either a retrosynthetic or a synthetic approach. In the first case, the target molecule is broken down into fragments which by some reactions lead to the target molecule. In the second case, starting materials are selected according to the structure of the target molecule, and then one checks that at least one synthetic path leads from these reagents to the target molecule. All basic chemical objects, such as atoms, bonds, molecules, and reactions, are described as frames in the knowledge representation language presented in an earlier section in this paper and in [Napoli 90]. Both synthetic and retrosynthetic approaches use classification-based processes described in the preceding section of this paper.

We briefly describe the retrosynthetic strategy in the following. The important chemical substructures which are potential reaction sites are organized in a graph called reactive structures graph. The second subsumption operation is used to build the graph: a structure *S1* subsumes a structure *S2* if *S1* is part of the description of *S2*. For example, the functional group *C=O* subsumes the formaldehyde molecule (two hydrogen atoms bonded to the carbon atom of *C=O*). The fact that the C=O group subsumes the formaldehyde molecule does not mean that the molecule inherits the properties of C=O, but that one can use chemical reactions attached to the group C=O to build the molecule.

To find methods to build a given target molecule *M*, the system tries to identify function groups in the reactive structures graph that are contained in the description of *M* by using a structural matching approach [Goel 89]. First, the system searches for the primitive reaction sites belonging to the target molecule. The primitive sites are simple structures comprised of a single bond, such as C=O. This first matching operation is therefore efficient, and guides the rest of the process: complex reaction sites are searched in the subgraphs of the reactive structures graph whose roots are the primitive reaction sites. Thus, the process for describing the target molecule in terms of reaction sites can be seen as a sequence of partial matching within the reactive structures graph.

Three special attributes, *schema+*, *schema-*, and *schema*, are attached to each reactive structure. They indicate the set of chemical reaction schemes which can build the structure. For a given structure *G*, schema+ introduces all reactions that build *G* but do not build its subsumers in the reactive structures graph. The attribute schema- introduces all reactions which do not build *G* but build its subsumers (this attribute plays a role analogous to shadowing for the inheritance relation). The value of schema is the complete list of reactions that build *G*. It is computed by the following formula: schema(G) = schema+(G) - schema-(G) U schema(g), for all g in the

ISSN: 2324-9773

subsumer set of G in the reactive structures graph. The computation of the list of the subsumers of *G* is analogous to the computation of the precedence list of a frame within the inheritance graph.

Given a target molecule *M*, the system tries to match part of the description of *M* to the reactive structures graph. These reaction sites, which we consider subsumers of *M* (according to the second view of subsumption) have schema attributes which contain reactions that can be applied to build *M*. When a method has been chosen and applied, the target *M* is broken into fragments. The process can be applied recursively to these fragments (new target molecules), until fragments of known starting materials are obtained.

The synthetic strategy for solving this problem is only partially treated and is still under study. Briefly, the system searches for basic reagents that subsume the target molecule. Basic reactions which are associated with these reagents, more precisely with the reagent family, are selected if they can lead from the reagent to the target molecule or a part of it.

BIBLIOGRAPHY

[Brachman 85] R.J. Brachman and J.G. Schmolze, An Overview of the KL-ONE Knowledge Representation System, Cognitive Science, 9(2):171-216, 1985.

[Cohen 84] B. Cohen and G.L. Murphy, Models of Concepts, Cognitive Science, 8(1):27-58, 1984.

[Ducournau 87] R. Ducournau and M. Habib, On some Algorithms for Multiple Inheritance in Object Oriented Programming, Proceedings of ECOOP'87, Paris, [Special issue of Bigre 54 or Lecture Notes in Computer Science 276), pages 291-300, 1987.

[Ducournau 89] R. Ducournau, Y3. Langage à objets. Version 3.22. SEMA GROUP, Montrouge, 1989.

[Finin 86] T.W. Finin, Interactive Classification: A Technique for Acquiring and Maintaining Knowledge Bases, Proceedings of the IEEE, 74(10):1414-1421, 1986.

[Goel 89] A. Goel and T. Bylander, Computational Feasibility of Structured Matching, IEEE Transactions on Pattern Analysis and Machine Intelligence, 11(12):1312-1316, 1989

[Laurenco 90] C. Laurenco and M. Py and A. Napoli and J. Quinqueton and B. Castro, Représentation de connaissances en synthèse organique à l'aide d'un langage à objets, New Journal of Chemistry, December 1990, (to be published).

[Lipkis 82] T. Lipkis A KL-ONE Classifier, in Proceedings of the 1981 KL-ONE Workshop, J.G. Schmolze and R.J. Brachman editors, Fairchild Technical Report no. 618, pages 126-143, 1982.

[Maes 87] P. Maes, Computational Reflection, Technical report, no. 87_2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[Masini 89] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. Les langages à objets, InterEditions, Paris, 1989.

[Napoli 90] A. Napoli, Using Frame-Based Representation Languages to Describe Chemical Objects, New Journal of Chemistry, December 1990, (to be published).

[Schmolze 83] J.G. Schmolze and T.A. Lipkis. Classification in the KL-ONE Knowledge Representation System In Proceedings of IJCAI'83, Karlsruhe, West Germany, pages 330-332, 1983.

[Touretzky 86] D.S. Touretzky, The Mathematics of Inheritance, Morgan Kaufmann Publishers Inc., Los Altos, California, 1986.

[Winston 84] P.H. Winston, Artificial Intelligence, Second Edition, Addison-Wesley, Reading, Massachusetts, 1984.

ISSN: 2324-9773