MULTIPLE INHERITANCE MECHANISMS IN LOGIC OBJECTS APPROACH BASED ON A MULTIPLE SPECIALISATION OF OBJECTS

Macaire Ngomo

Laboratoire d'Analyse Numérique, d'Informatique et Applications (LANIA), Faculté des Sciences et Techniques (FST) de l'Université Marien NGOUABI, Brazzaville, République du Congo

CM IT CONSEIL – Département Ingénierie, Recherche, Innovation – 32 rue Milford Haven 10100 Romilly sur Seine, France

IMAGINE TECHNOLOGY, Département Ingénierie, Brazzaville, République du Congo

ABSTRACT

This study takes place within the framework of the representation of knowledge by objects and within the framework of our work on the marriage of logic and objects. On the one hand, object-oriented programming has proved to be appropriate for the construction of complex software systems. On the other hand, logic programming is distinguished by its declarative nature, integrated inference and well-defined semantic capabilities. In particular, inheritance is a refinement mechanism whose mode of application leaves a number of design choices. In the context of this marriage, we describe the semantics of multiple inheritance in a non-deterministic approach based on multiple specification of logical objects. We also describe the conceptual choices for integrating multiple inheritance made for the design of the OO-Prolog language (an object-oriented extension of the Prolog language respecting logical semantics).

KEYWORDS

Object-Oriented Logic Programming, Object-Oriented Representation, Multiple Inheritance, Multi-Point of View, Semantics of Multiple Inheritance

1. INTRODUCTION

Inheritance is a refinement mechanism whose mode of application leaves a number of design choices. In this article, we describe the semantics of inheritance (Cardelli, 1984) in a non-deterministic approach as well as the conceptual choices of integration of monotonous multiple inheritance made for the design of the OO-Prolog language (Ngomo, 1996) as well as its application to the dynamic classification by multiple specialisations of logical objects. Our work concerns the multiple and evolutionary representation of objects that supports reasoning by classification (Capponi & Chaillot, 1993; Dekker, 1993; Drews, 1993; Euzenat, 1993; Marino & al., 1990; Marino & al., 1990; Masini & al., 1989; Mac Gregor & Burstein, 1991; Mac Gregor & Brill, 1992; Napoli, 1992; Rechenmann, 1984). This representation must therefore allow a dynamic classification of logical objects and follow classificatory reasoning. The inheritance management model of the OO-Prolog language is based on the non-determinism of logic programming, on explicit naming and on the concept of full attribute naming which allows conflicts to be resolved before they arise. The OO-Prolog language adopts a dynamic inheritance for both attributes and methods.

2. INHERITANCE IN THE OBJECT-ORIENTED APPROACH

The paradigm of object-based programming, born with Smalltalk (Goldberg & Robson, 1987) at the end of the 1970s, gives a great power of expression, ease of maintenance and reusability superior to other paradigms: imperative, functional (example with LISP Steele, 1990; Sterling & Shapiro, 1990) or logical (example with PROLOG (Sterling & Shapiro, 1990), etc. However, it requires a greater capacity for abstraction than

imperative or functional programming in order to choose the "objects" to be reified and to define inheritance and composition between classes in a meaningful and coherent way. The main dimensions of the object paradigm which are classification, inheritance which introduces the notions of generalisation and specialisation, encapsulation and polymorphism (generic functions) were brought together for the first time in Smalltalk 76 (Goldberg & Robson, 1987), although the ideas of class and instance, and inheritance had matured with SIMULA (Doma, 1988). Classes were seen as objects, created by metaclasses, in the object languages created above Lisp, then in Smalltalk 80 (Goldberg & Robson, 1987) and this vision was taken up again in Java where everything is an object, the elements of world representation, the elements of graphical interfaces, but also the elements of the language like functions, classes, events, errors and exceptions. Composition was later added as an autonomous dimension with UML and is present in modern languages such as Java.

2.1 Inheritance

Inheritance is a mechanism for sharing information by factoring members. When several classes have common characteristics, it is possible to create a more general classifier that groups together these structure (classes) or behaviour (interface) properties. Inheritance makes it possible to infer all the members of a class that are not explicitly given there. This mechanism of inference comes back to an algorithm for browsing the class graph according to a defined strategy. Multiple inheritance is an extension to the simple inheritance model where one class is allowed to have several parent classes in order to model multiple generalization. An object can be considered from several points of view. However, the use of multiple inheritance is not without its problems. For example, if the two base classes have attributes or methods with the same name, there are naming collisions that need to be resolved. If inheritance causes a conflict of methods, a conflict resolution strategy, i.e. a choice or combination procedure as in CLOS (Daniel & al., 1992; Keene, 1989) should be used. We will come back to this dimension to describe the conceptual choices of integration of multiple inheritance made for the design of the OO-Prolog language and the strategies for resolving inheritance conflicts.

2.2 Inheritance Semantics

Almost all object languages implement a notion of inheritance between classes. The principle is to specialize and factorize. However, inheritance is a refinement mechanism whose mode of application leaves a certain number of design choices. To do this, we are faced with two design choices: the semantics of inheritance [Cardelli, 1984] and the path strategy of the inheritance graph. In this section, we come back to this concept of inheritance in order to describe its semantics as well as the choices that were retained for the conception of the OO-Prolog language. The traditional definition of inheritance presupposes non-monotonous semantics in the composition of the different inherited classes. Several languages and models are based on this inheritance model. In these languages, the semantics of inheritance is non-monotonic. Generally, these languages use the same strategies as those of common object languages such as the linearization of the classes of the inheritance graph. Examples are ObjVProlog (Malenfant, 1990]) and Prolog++ (Moss, 1994). Others support multiple inheritance and offer no means of resolving conflicts (e.g. the systems of Kowalski (Kowalski, 1979a; Kowalski, 1979b) and Zaniolo (Zaniolo, 1984)). Gallaire (1986), Leonardi and Mello (1988) propose, in object-oriented logic programming, to replace non-monotonous semantics by monotonous semantics where, by backtracking, one would explore vertically all the definitions, from the subclasses to the superclasses. This approach is interesting from the point of view of first-order logic, which is monotonous in nature. However, it poses a major problem. Indeed, if inheritance is used to build on the basis of another class, which supports the idea of monotonous semantics, it is also used to differentiate behaviours. It often happens that an entity is modelled by a class, saying: my instances will be like those of such and such a class (inheritance) except for such and such behaviour (differentiation). This last interpretation therefore requires non-monotonous semantics. This necessity to have a way to reintroduce non-monotonous semantics of inheritance has led Gandilhon (1987) to propose a new form of cut to prevent the backtracking on definitions in inherited classes. He calls this cut "cut_inheritance". Monotonous semantics provides a solution from the point of view of first-order logic programming. However, OO-Prolog adopts non-monotonic inheritance semantics because it is more common in object-oriented programming languages. For the design of OO-Prolog, we have retained the non-monotonous semantics of inheritance for two main reasons:

- because the traditional definition of inheritance assumes non-monotonic semantics in the composition of the different inherited classes
- because it is the most common in object languages and is necessary in many cases to differentiate the behavior of objects.

3. TECHNIQUES FOR RESOLVING INHERITANCE CONFLICTS

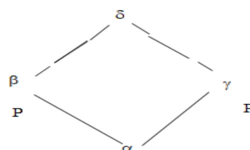

Figure 1. Primitive scene

Inhitance is a mechanism for both hierarchical and deductive information sharing, defined on a set of objects partially ordered by a specialisation relationship. Each of these classes has properties which are the object of inheritance. As a first approximation, these properties have values and a name (or selector). Multiple inheritance allows a more flexible modeling of an application by avoiding the multiplication of useless classes. On the other hand, this form of inheritance can introduce conflicts. The problem of conflicts falls within the general framework of Figure 1 taken from (Ducournau, 1993; Napoli & Ducournau, 1992).There is no universal technique for resolving these kinds of conflicts and there are a wide variety of techniques for resolving them. Different views on how to resolve them are often contradictory. In software engineering, the risks of error and confusion must be avoided at all costs: conflicts are generally prohibited because they are incompatible with a programming framework based on rigour and reliability. In artificial intelligence, multiple inheritance is a natural and indispensable principle for modelling real-world situations and entities. We describe below the common techniques (Booch, 1992; Bouché, 1994; Meyer, 1987; Meyer, 1990).

3.1 Conflict Resolution by Mistake

Error-based conflict resolution occurs when the semantics of the language consider the collision to be illegal and cause an error in compiling the inheriting subclass.

3.2 Conflict Resolution by Equivalence

We speak of conflict resolution by equivalence when the semantics of language consider the same name introduced by different classes as referring to the same field.

3.3 Conflict Resolution by Renaming

Conflict resolution by renaming occurs when the semantics of the language consider the same name introduced by different classes as referring to distinct fields, and thus duplicate the renamed components. The expressions "conflict resolution by duplication" and "conflict resolution by renaming" are synonymous. The Eiffel language uses this principle (Bouché, 1994; Meyer, 1987).

3.4 Conflict Resolution by Qualification

We speak of conflict resolution by qualification when the semantics of language require that all references to the selector fully qualify the source of its statement. In C++, for example, the attribute name includes the name of the overclass, so references to the name fully qualify the source of its declaration.

3.5 Conflict Resolution by Points of View

Here is an object-oriented description of the Computer with a technical and an accounting interpretation. In the example below, inheritance conflicts over the Duration and Priority attributes are handled by viewpoints in OBJLOG (Chouraqui & Dugerdil, 1988; Dugerdil, 1988; Dugerdil, 1991). Let us imagine the Computer class (see Figure 2). This class inherits the Accounting Service and Computer Workshop classes. The Accounting Service class will have a Lifetime attribute (depreciation period) and the Technical Service class will also have the Lifetime attribute (warranty period). When you want to access this attribute, you will have to specify by some means or other if you want to access its value from a "technical" or "accounting" point of view.

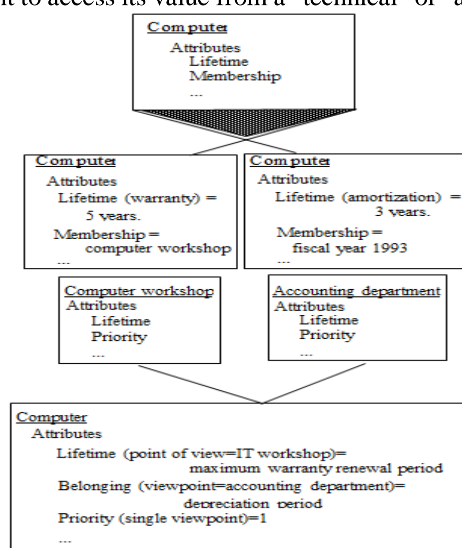

Figure 2. Points of view

"A point of view is an interpretation of all or part of the data of a class corresponding to an abstraction of the real world" (Bouché, 1994). A class may therefore have several points of view. The sum of these points of view will be called perspective. "A perspective is a composite class representing different interpretations (points of view) of the same abstraction of the real world" (Bouché, 1994). We will speak of conflict resolution by points of view when the semantics of the language use the modelling of perspective classes decomposed by the delimitation of points of view. This concept is fundamental in the problematic of knowledge representation (Vogel, 1988) where different types of knowledge do not have the same meaning in different domains of discourse. The OBJLOG language, for example, defines a mother class as a point of view for a daughter class. Unlike CLOS, which resolves possible conflicts by using a precedence list, OBJLOG enshrines the notion of point of view. The conflict resolution algorithm will reason by difference or equivalence of points of view.

3.6 Conflict Resolution by a Combination of Methods

The combination of methods aims, when sending a message, to combine the execution of different methods of the same object. These methods which have the same selector are in call conflict. This technique, used for example in the FLAVORS system, consists of labelling the methods in order to determine a certain sequence. In the KEE language, these labels aim at managing specialisation to avoid arbitrary masking of the methods code (overloading) or more generally conflicts in multiple inheritance (Bouché, 1994). In this case, a parameterization of the path of the inherited classes is given by the combination. This principle of method combination is at the basis of the generic functions introduced in the CLOS language (Daniel & al., 1992; Keene, 1989). We speak of conflict resolution by method combination when the semantics of the language use the notion of method labelling (daemon) to allow a certain chaining, moreover the combination provides a parameterisation of the path of the inherited classes.

3.7 The Path of the Inheritance Graph

In many languages, inheritance conflicts are resolved by defining an order in which outliers will be examined to find the property definition that will be used to respond to a message. Classically, this is equivalent to defining a total or partial order in the inheritance graph or in the sub graph whose source is the instantiation class of the object that receives the message. If the searched property is located at different places in the hierarchy, the first class found by the execution of the path algorithm will be selected; hence the importance of knowing the algorithm used during programming in order to predict the result. Here the direction of the graph will play a role in resolving the conflict since it will, to a certain extent, specify the priorities of the classes. Linear techniques have the major disadvantage of systematizing the treatment of each conflict, without taking into account the semantics of the properties involved. As Masini (Masini & al., 1989) points out, conflict resolution can only be reliable if it takes into account the knowledge related to the application. Systematically applying a default solution cannot therefore correctly resolve each case. The algorithms used in the graph must therefore be taken into account according to the nature of the problems to be solved (Bouché, 1994). Certain modes of conflict resolution prevent this arbitrary choice, dictated by the chronology of class specialization.

4. INHERITANCE MECHANISMS IN ORIENTED-OBJECT LOGIC APPROACH

The OO-Prolog is one of the many hybrid languages resulting from work on the integration of object-oriented programming paradigms and logic programming paradigms (Ait-Kaci & Podelski, 1991; Ait-Kaci & Podelski, 1993; Andreoli & Pareschi, 1992; Bowen & Weinberg, 1985; Chen & Warren, 1988; Conery, 1988; Gandriau, 1987; Davison, 1993; Ishikawa & Tokoro, 1987; Ishikawa & Jungclaus, 1993; Malenfant, 1990; Moss, 1994; Ngomo, 1996; Mac Gregor & Burstein, 1991). It supports multiple inheritance with non-monotonic semantics. To resolve inheritance conflicts, we adopt a solution based on non-deterministic resolution, the notion of viewpoint and the concept of full attribute name. For many common object languages, a default graph traversal strategy is required. Linear strategies remain, for the moment at least, the best compromise (Masini & al., 1989). For some, they are currently the only acceptable techniques (Ducournau, 1993; Napoli & Ducournau, 1992). However, three reasons lead us to propose a non-linear, non-deterministic approach for object-oriented logic programming. Firstly, as Masini (Masini & al., 1989) points out, there is probably no universal, ideal linear strategy that is satisfactory in all cases. Secondly, linear techniques have the major drawback of systematising the treatment of each conflict, without taking into account the semantics of the data involved. Finally, the possibility offered by Prolog to explore, by backtracking, all possible alternatives, allows, in case of ambiguities, to consider an object with all its points of view (without any discrimination). OO-Prolog adopts a dynamic inheritance for both attributes and methods. However, attribute inheritance and method inheritance are treated differently.

4.1 Attribute Inheritance

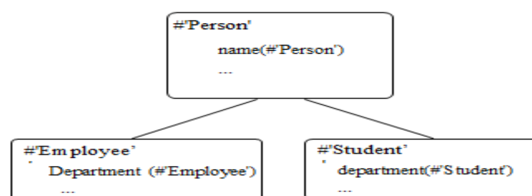

Figure 3. Full name of an attribute in OO-Prolog

For the choice of the inheritance model of the OBJLOG language, Dugerdil and Chouraki hypothesised that the conflicting attributes do not have the same semantics (Chouraqui & Dugerdil, 1988; Dugerdil, 1988; Dugerdil, 1991). We take up some of OBJLOG's ideas and retain this hypothesis in particular to provide the means to resolve name conflicts before they arise. In OO-Prolog, attribute name conflicts are resolved by the concept of full name (Escamilla, 1990). If an attribute is defined in a class, its full name is the term whose

functor is equal to the attribute name and whose only argument is the definition class. This means that two attributes with the same name but not having the same origin (definition class) have different full names and are considered semantically different. This is the case here for the 'department' attributes defined in the classes #'Employee' and #'Student' (Figure 3). As we have already seen, an attribute is represented by a Prolog term of arity one. Its argument corresponds to the point of view that determines the interpretation of the attribute: <name>(<interpretation>). Each attribute inherited from an overclass therefore has a different interpretation from the others. A class then inherits all the attributes of its upgrades. Two attributes are homonymous if they have the same name and if the intersection of their labels is empty (for example, department(#'Employee') and department(#'Student') are homonymous). Conversely, two attributes are different if their names are different (for example, name(#'Person') and age(#'Person') are different). As in OBJLOG, we define a mother class as a point of view for a daughter class. Thus we can use the inheritance relation to introduce the notion of disjunctive interpretation of an attribute at the level of a class C, i.e. the set of interpretations of the attributes of the same name in the subgraph of C. It corresponds to the set noted {c1,...,cn}, where ci are classes, maximum lower bounds for this attribute at the level of class C.

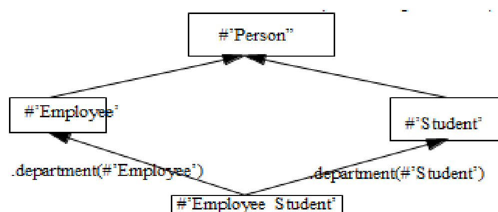

Figure 4. Interpretation of an attribute

In the context of Figure 4, the disjunctive interpretation of the 'department' attribute at class level #'Employee_Student' is {#'Employee',#'Student'}. The disjunctive interpretation of an attribute at the level of its definition class is the singleton composed of this same class. For example, the disjunctive interpretation of the department attribute at the class level #'Employee' is the singleton {#'Employee'}. Thus, when in a method call the interpretation of an attribute is a free variable, it is unified with each of the elements of the disjunctive interpretation of this attribute at the level of the current class. Let O therefore be an instance of the class #' Employee_Student ', having for study department "La Seine-Maritime" and for work department "La Haute-Seine". The processing of the following request is done as follows:

- first find the disjunctive interpretation of the "department" attribute at the level of the current class, here Employee _Student: {#' Employee',#'Student'},
- using backtracking, instantiate the variable Int with each of the elements of this set and calculate the value of the attribute corresponding to each interpretation.

One of its subclasses can be specified as in the following example. In this case, the value of the attribute is calculated in the same way, considering as the disjunctive interpretation of this attribute at the level of the subclass specified when calling the method.

4.2 The Inheritance of Methods

In this section we discuss one aspect of inheritance which is the inheritance of behaviour. We are in the most general case, that of multiple inheritance. Behaviour inheritance is a synthesis of the consequences of the inheritance relation at the level of methods; it describes the evolution of the behaviour of classes through user-defined inheritance links. In OO-Prolog, method inheritance is also dynamic but managed differently by three complementary strategies which can be combined dynamically.

4.3 The Non-Deterministic Strategy

In order to consider an object with all its points of view, OO-Prolog uses a partial order with backtracking in the case of remaining ambiguities. By default, sending a message activates all methods in conflict, taking advantage of the backtracking performed by the Prolog interpreter in his exhaustive search for solutions to a query.

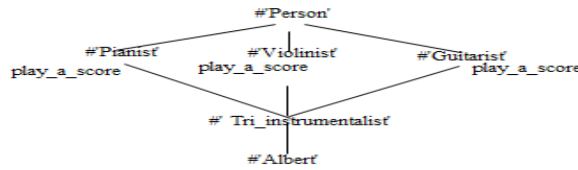

Figure 5. Points of view of #'Albert'

For example, in figure 5, #'Albert' designates an instance of the class #'Tri-instrumentalist' which itself inherits three classes: Pianist, #'Violinist', #'Guitarist'. In each of these classes the method play_a_score is defined. If Albert is asked to play a score by sending him the following message "#'Albert' <- play_a_score", which instrument will use #'Albert' to play his score? In a linear approach in which classes are given priority, Albert will consider the class with the highest priority and therefore use the instrument corresponding to that class by default. For example, in CLOS it will be the Pianist class. In OO-Prolog, this message is transformed into or logical on the maximum lower bounds of this method at the level of the class #'Tri-instrumentalist' ({#'Pianist', #'Violinist', #'Guitarist'}). This prevents an arbitrary choice dictated by the chronology of class specialisation and thus prevents the object from being questioned from all points of view (or in all its aspects). We can multiply examples of this kind. In the context of Figure 6, sending the message department(D) to the object #'Paul' is equivalent to #'Paul' as #'Employee' or #'Paul' as #'Student'. Thus, by default, OO-Prolog does not deal with method inheritance conflicts. Sending a message activates all the conflicting methods, taking advantage of the feedback provided by the Prolog interpreter in his exhaustive search for solutions to a query. Thus, while in the monotonous approach, backtracking is used to introduce monotonous inheritance semantics (Figure 7), we use it here to avoid introducing a horizontal order between classes. This makes it possible to consider an object with all its points of view, without any discrimination. In classical approaches, a choice is made, with no possibility of going back. In OO-Prolog, backtracking allows the application of all conflicting methods (Figure 8).

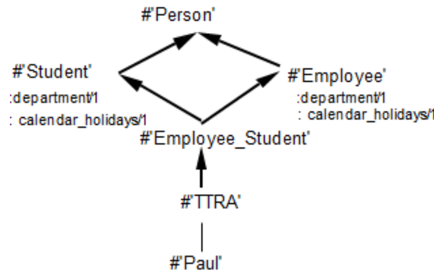

Figure 6. Student and Employee: which department/1 instance O uses at the TTRA level?

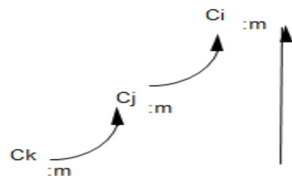

Figure 7. Vertical backtracking

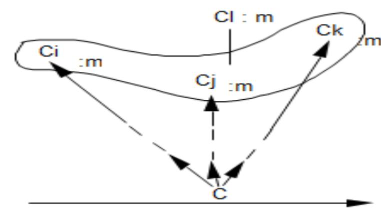

Figure 8. Horizontal backtracking

By default, the general rule is therefore that sending a message triggers all possible methods, taking advantage of the feedback provided by the Prolog interpreter. For example, in the context of Figure 6, sending the message department(D) to the object #'Paul' of the TTRA class is equivalent to or logical and is dealt with by exploring conflicting classes by backtracking. This strategy is in our opinion more general than a classical non-monotonous linear strategy such as Pclos, P1, etc. in that any solution obtained using such a linear strategy can also be a solution of this approach. For example, in the context of Figure 6, P1 and Pclos consider the class #'Student' as having a higher priority than the class #'Employee'. The object will therefore respond to the department(D) message as a student and eventually return to its study department.

4.3.1 Linear Strategy

A message of the form "O <-- Message" is processed using a predefined linear extension algorithm. Linear strategies are to be taken into account according to the nature of the problems to be solved. They do not always give the same result. The user must therefore be given the possibility to introduce his own strategies or to use several existing strategies (Pclos, P1, Pflavors, etc.). The solution currently adopted in OO-Prolog consists in making available to the programmer several path strategies that he can use according to his needs. By default it is the inversion or P1 route strategy that will be considered by the system. A simplified version of the inversion algorithm consists in removing the nodes from the graph in the order of stacking the deep path first, without masking the nodes already visited: the result therefore contains several occurrences of certain nodes. The resulting list is then browsed in reverse, removing as it goes along the elements already encountered at least once. In this way, only the last occurrence of each element in the initial list is kept in the final list.

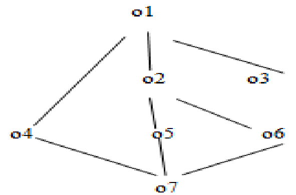

Figure 9. Example of an inheritance graph

Let us consider the graph in Figure 9 and calculate the priority list of o7 using this algorithm. The list provided by the depth path first is as follows: o7, o4, o1, o5, o2, o1, o6, o2, o1, o3, o1. The priority list obtained after removing duplicates is as follows: o7, o4, o5, o6, o2, o3, o1. The definition of a linear strategy is done by defining the predicate lookup (Class,Precedence,LookupName) where Class is the class at which the graph starts and Precedence is the precedence list of the Class class. The LookupName parameter is the name of the strategy. Thus, it is possible to define several independent linear strategies and to use them all in the same application. The primitive set_lookup then dynamically sets the strategy to be used: set_lookup(S), S being the strategy to be set. For example, if the user defines the strategy of CLOS, to fix it, just execute the goal : set_lookup(pclos). The primitive get_lookup(S) unifies variable S with the name of the current strategy. This assignment is temporary and defeated by backtracking.

4.3.2 The Explicit Designation

It consists in explicitly designating a class to which a method belongs. It is a tool made available to the user and allowing him/her to have greater control over the inheritance mechanism. By explicitly designating the class of origin of a property, it is thus possible to make certain choices "by hand", thanks to a horizontal masking of the other classes. A designation may be incomplete. This is the case when the designated class is not the one in which the property is defined, but one of its superclasses. In this case, the basic strategy will be used, starting from the designated class. The explicit designation is introduced by the ":"/2 operator:

<(<object> <- (<class>):<message>

Still in the context of Figure 6, the application allows you to consider the object O, a direct instance of the class #'Employee_Student', as a direct instance of the class #'Employee' and to hide horizontally the department/1 method defined in the class #'Student'.

(a) Explicit multiple designation

In OO-Prolog, the explicit designation can be multiple, i.e. several classes can be designated. Although the designated classes are considered in this order, it is not of great importance since, regardless of the order given, the result is the same. Thus, we can also write:

?- D <- ([#'Swimming_Bird', #'Flying_Bird']) :mode(Mode).

(b) Explicit designation and masking

When a class is explicitly designated, a control mechanism makes it possible to check that the principle of vertical masking is respected.

(c) Designation and path of the inheritance graph

It is also a means of reducing the complexity of the methods in the inheritance graph, since it consists of making a jump to the designated class and consequently reducing the method search graph.

5. CONCLUSION AND PERSPECTIVES

We have described a new model for handling multiple inheritance and its application in the context of object-oriented logic programming. The logic approach has many advantages over traditional methods. It brings a lot of flexibility on the handling of inheritance. Our method of handling inheritance avoids an arbitrary choice dictated by the system. It is even possible to combine different strategies depending on the problem to be treated. This vision of inheritance allows us to implement more easily classification mechanisms based on multiple specialization. Such mechanisms combined with the delay mechanism allow to freeze all the classification constraints and to trigger them only when the missing data are available.

REFERENCES

Ait-Kaci & Podelski, (1991) Ait-Kaci, H. & Podelski, A. "Towards a Meaning of LIFE". Proc. of the Thirsd Int'l Conf. on Programming Language Implementation and Logic Programming, Lectures Notes in Comp. Sciences, Passaü, Aug. 1991.

Ait-Kaci & Podelski, (1993) Ait-Kaci, H. & Podelski, A. "Towards a Meaning of LIFE". Journal of Logic Programming, 16:195-234, 1993.

Andreoli & Pareschi, (1992) Andreoli J.M., Pareschi R., "Linear objects: A logic framework for open system programming", In A. Voronkov, editor, Inter. Conference on Logic Programming and Automated Reasoning LPAR'92, pp 448 450, St. Petersburg, Russia, July 1992.

Daniel & al., (1992) Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E.Keene, Gregor Kiczales and David A. Moon. “Common Lisp Object System specification”, ACM SIGPLAN Notices, 1988

Booch, (1992) Booch G. “Object Oriented Design with applications” The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1992.

Bouché, (1994) Bouché M., "La démarche objet. Concepts et outils.", AFNOR, 1994.

Bowen & Weinberg, (1985) Bowen, K.A. et Weinberg, T. A Meta Level Extension of Prolog, IEEE Inttl Symp. on Logic Prog. 'B5 (1985), pp.48 53.

Cardelli, (1984) Cardelli L., A semantics of Multiple Inheritance, LNCS, vol.137, Springer-Verlag, pp.51-67, 1984.

[Capponi & Chaillot, (1993) Capponi C., Chaillot M., Construction incrémentale d’une base de classes correcte du point de vue des types, Actes Journée Acquisition-Validation-Apprentissage, Saint-Raphael, 1993.

Chouraqui & Dugerdil, (1988) Chouraqui E., Dugerdil Ph., Conflict olving in a Frame-like Multiple Inheritance System, ECAI, Munich, pp.226-232, 1988.

Chen & Warren, (1988) Chen W. and Warren D. S.. Objects as intensions. In Logic Programming: Proc. 5th Int'l Conf. and Symp., Seattle, WA, USA, 15 19 Aug 1988, pages 404 19. The MIT Press, Cambridge, MA, 1988.

Conery, (1988) Conery J., Logical Objects. Proc. of the Fifth Int'l Conf. on Logic Prog. , p.p. '20-443, 1988.

Davison, (1993) Davison, A. "A Survey of Logic Programming-based Object Oriented Languages". In Research Directions in Concurrent Object-Oriented Programmaing. The MIT Press, Cambridge, MA, 1993.

Dekker, (1993) Dekker L., « Frome : représentation multiple et classification d'objets avec points de vue », Thèse de doctorat en Sciences appliquées, Sous la direction de Gérard Comyn. Soutenue en 1994, à Lille 1.

Doma, (1988) Doma, A. "Object-Prolog: Dynamic Object-Oriented Representation of Knowledge". In T. Henson, editor, SCS Multiconference on Artificial Intelligence and Simulation: The Diversity of Applications, pages 83-88, San Diego, CA, Feb. 1988.

Drews, (1993) Drews O. M.. Raisonnement classificatoire dans une représentation à objets multi-points devue. Interface homme-machine [cs.HC]. Université Joseph-Fourier - Grenoble I, 1993. Français. tel-00005133

Ducournau, (1993) Ducournau R., Héritages et représentations, Mémoire, Diplôme d’Habilitation à diriger des recherches, spécialité : Informatique, Université Montpellier II, 1993.

Dugerdil, (1988) Dugerdil P., Contribution à l’étude de la représentation des connaissances fondée sur les objets. Le langage OBJLOG. Thèse de l’Université d’Aix-Marseille II, 1988.

Dugerdil, (1991) Dugerdil P., Inheritance Mechanisms in the OBJLOG language : Multiple Selective and Multiple Vertical with Points of View in Inheritance Hierarchies in Knowledge Representation, M.Lenzerini, D.Nardi and M.Simi (éd.), John Wiley & Sons Ltd., pp.245-256, 1991.

Escamilla, (1990) Escamilla J., JEAN P., Relationships in an Object Knowledge Representation Model, Proceedings IEEE. 2nd Conference on Tools for Artificial Intelligence, Washington D.C. USA, pp.632-638, 1990.

Euzenat, (1993) Euzenat J., Classification dans les représentations par objets : produits de systèmes classificatoires, Rapport interne, Equipe SHERPA, INRIA, 1993.

Gallaire, (1986) [Gallaire, 1986] Gallaire, H. "Merging Objects and Logic Programming: Relational Semantics, Performance and Standarization". In Proc. AAAI'86, pp.754-758, Philadelphia, Pensylrania, 1986.

Gandilhon, (1987) Gandilhon T. "Proposition d'une extension objet minimale pour Prolog.", Actes du séminaire de Programmation logique, Trégastel (mai 1987), pp. 483-506.

Gandriau, (1987) Gandriau, M. "CIEL: classes et instances en logique". Thèse de Doctorat, ENSEEIHT 1988, 151p.

Goldberg & Robson, (1987) Goldberg, A. and Robson, D. "Smalltalk-80: The language and its implementation". Addison-Wesley, 1983.

Ishikawa & Tokoro, (1987) Ishikawa, Y. et Tokoro, M. Orient84/K: An ObJect Orlented Concurrent Programming Language for Knowledge Representation, Object Oriented Concurrent Programming (1987), W 159 198.

Ishikawa & Jungclaus, (1993) Ishikawa, Y. R. Jungclaus. Logic Based Modeling of Dynamic Object Systems. PhD thesis, Technical University Braunschweig, Germany, 1993.

Keene, (1989) Sonja E. Keene, "Object-Oriented Programming in Common Lisp : a Programmer's Guide to CLOS", Addison-Wesley, 1989.

Kowalski, (1979a) Kowalski, R. "Algorithm = Logic + Control", Comm. ACM 22, 7 (1979), 424-436.

Kowalski, (1979b) Kowalski, R. "Logic for problem solving". North-Holland, Amsterdam, 1979.

Leonardi & Mello, (1988) Leonardi L. and Mello P., "Combining logic- and object-oriented programming language paradigms", in Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences. Volume II: Software track, Kailua-Kona, HI, USA, 1988 pp.376-385. doi: 10.1109/HICSS.1988.11828

Malenfant, (1990) Malenfant, J. "Conception et Implantation d'un langage de programmation intégrant trois paradigmes: la programmation logique, la programmation par objets et la programmation répartie". Thèse de PhD, Univ. de Montréal, Mars 1990.

Marino & al., (1990) Mariño O., Rechenmann F., Uvietta P. Multiple perspectives and classification mechanism in object-oriented representation, 9th ECAI, pp.425-430, Stockholm 1990.

Marino & al., (1990) Marino O., Classification d'objets composites dans un système de représentation de connaissances multi-points de vue, RFIA'91, Lyon-Villeurbanne, pp. 233-242, 1991.

Masini & al., (1989) Masini G., Napoli A. Colnet D. Leonard D., Tombre K., Les langages à objets. Intereditions, Paris, 1989.

Mac Gregor & Burstein, (1991) Mac Gregor R.M., Burstein M.H. Using a Description Classifier to Enhance Knowledge Representation, IEEE Expert Intelligent Systems and Applications, juin, 1991.

Mac Gregor & Brill, (1992) Mac Gregor R.M., Brill D., Recognition Algorithms for the LOOM Classifier, AAAI, San José, CA, juillet, pp.774-779, 1992.

Meyer, (1987) Meyer B."Eiffel: Programming for reusability and extendibility", ACM SIGPLAN Notices, 22(2):85-94, 1987.

Meyer, (1990) Meyer B. "Conception et programmation par objets, pour le génie logiciel de qualité", InterEditions, Paris 1990.

Moss, (1994) Moss C., "Prolog++: The Power of Object-Oriented and Logic Programming", Addison-Wesley, 1994.

Napoli, 1992 Napoli A., « Représentations à objets et raisonnement par classification en intelligence artificielle », Thèse de doctorat en Informatique. Soutenue en 1992, au CRIN - Centre de Recherche en Informatique de Nancy, France.

Napoli & Ducournau, (1992) Napoli A., Ducournau R., Subsumption in Object-Based Representations,Proceedings ERCIM Workshop on theoretical and practical aspects of knowledge representation, (rapport ERCIM 92-W001) pp1-9, Pisa (IT), 1992.

Ngomo, (1996) Ngomo M. "Intégration de la programmation logique et de la programmation par objets : étude, conception et implantation". Thèse de Doctorat d'Informatique, Université de Rouen - INSA Rouen, Décembre 1996.

Vogel, (1988) Vogel C., Génie Cognitif, Collection Sciences cognitives, MASSON, PAris, pp.97 1988.

Steele, (1990) Steele G. L. "Common Lisp : the language" second edition, Digital Press, 1990.

Sterling & Shapiro, (1990) Sterling, L. et Shapiro, E. "L'Art de Prolog". MASSON 1990.

Zaniolo, (1984) Zaniolo, C. "Object-Oriented Programming in Prolog". In Proc. of the IEEE Internatial Symposium on Logic Programming, pp. 265-270, Atlantic City, New Jersey, 1984.

Rechenmann, (1984) Rechenmann F., Bensaid A. et Granier D., SHIRKA: des systèmes experts centrés-objet. Actes 4ièmes journées internationales sur les systèmes-experts et leurs applications, actes polycopiés, Avignon, FR, 1984.