International Journal of Computer Trends and Technology
ISSN: 2231 – 2803 / https://doi.org/10.14445/22312803/IJCTT-V68I11P104

Volume 68 Issue 11, 24-42, November 2020
© 2020 Seventh Sense Research Group®

Original Article

Scalable multiple representation and dynamic classification by multiple specialization of objects in OO-Prolog

Macaire Ngomo[1]

CM IT CONSEIL – Engineering and Innovation Department – 32 rue Milford Haven 10100 Romilly Sur Seine (France)

Received Date: 30 September 2020
Revised Date: 13 November 2020
Accepted Date: 14 November 2020

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 constructing 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 several design choices. In the context of this marriage, we describe the semantics of multiple inheritances in a non-deterministic approach, the conceptual choices of integration of multiple inheritances made for the design of the OO-Prolog language (an object-oriented extension of the Prolog language respecting logical semantics) as well as its application to multiple evolutionary representations that support classificatory reasoning and to dynamic classification by multiple specifications of logical objects.*

Keywords - *Object-oriented logic programming, object-oriented representation, multiple inheritances, multi-point of view, classificatory reasoning.*

I. 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 [11] [12] in a non-deterministic approach as well as the conceptual choices of integration of monotonous multiple inheritances made for the design of the OO-Prolog language (an object-oriented extension of the Prolog language respecting logical semantics) [72] [73] [74] [75] [76] [77] [78] [79] as well as its application to the dynamic classification by multiple specializations of logical objects. Our work concerns the multiple and evolutionary representation of objects that supports reasoning by classification [68] [21] [24] [17] [14] [30] [52] [53] [54] [55] [57] [NEBEL 90] [QUI93] [58] [35]. This representation must, therefore, allow a dynamic classification of logical objects and follow classificatory reasoning. Reasoning by classification consists of finding the most specialized class or category to which an object belongs and retrieving knowledge related to this location.

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. This is a difference with classical models such as the ObjVLisp model from which it was inspired. Let us recall that ObjVLisp makes a static inheritance of the instance variables, which results in the flattening of the inheritance graph regarding the state of an object. The result is that an object in ObjVLisp is a vector of instance variables where all inheritance information has disappeared.

II. THE OBJECT PARADIGM AND ITS DIMENSIONS

The paradigm of object-based programming, born with Smalltalk [37] at the end of the 1970s, has become very popular: object-based languages, object-based representations in artificial intelligence, object databases, object-based design in software engineering, etc. The paradigm of object-based programming is now being used in many different fields. It gives great power of expression, ease of maintenance, and reusability superior to other paradigms: imperative (example with C), functional (example with LISP [89] [90] [90]) or logical (example with PROLOG [88] [91] [90]), etc. However, it requires a greater abstraction capacity than imperative or functional programming to choose the "objects" to be reified and define inheritance and composition between classes in a meaningful and coherent way.

This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)

The main dimensions of the object paradigm which are classification, inheritance which introduces the notions of generalization and specialization, encapsulation and polymorphism (generic functions), were brought together for the first time in Smalltalk 76 [37], although the ideas of class and instance, and inheritance had matured with SIMULA [23]. Classes were seen as objects, created by metaclasses, in the object languages created above Lisp, then in Smalltalk 80 [37]. This vision was taken up again in Java, where everything is an object, the elements of world representation, the elements of graphical interfaces, and the elements of the language like functions, classes, events, errors, and exceptions. The composition was later added as an autonomous dimension with UML and is present in modern languages such as Java.

A. Encapsulation

In the object paradigm, encapsulation concerns the grouping of variables and functions into classes and the grouping of classes and interfaces into packages. Classes, functions, and packages are also namespaces that ensure uniqueness within the names of the elements composing them. From the outside, it may be necessary to prefix the names of imported public elements by the name of the class or package from which the referenced element comes (or by this or by super). Encapsulation ensures the grouping in the same elements (classes or packages) of lower-level elements strongly linked. It ensures the protection and partial visibility of the elements outside. Encapsulation ensures the independence between a class's layout, a function, a package, and how it is presented concerning the other objects using it. The public presentation of an element ensures that a contract will bind that element about what it does, but not how it does it, which is the responsibility of its implantation. Therefore, it can be changed without affecting the operation of the other elements that use it, for example, to change internal variables or the algorithms used. The encapsulation and access levels (private, public, etc.) give rise to the reusability of software elements and software evolution.

B. Inheritance

The organization of classes in specialization hierarchies makes it possible to create complex classes from more general classes by refining the general description. A subclass is built from another class by adding members or restricting members existing in the other class. The mechanism by which a class retrieves information inherited from its superclasses is called inheritance. Inheritance is, therefore, a mechanism for sharing information by factoring in members. Inheritance between classes allows the reuse of the structures or behaviours introduced, and facilitating updating, avoiding duplication of information. When several classes have common characteristics, it is possible to create a more general classifier that groups together these structures (classes) or behaviour (interface) properties. It reduces the need to specify redundant information and simplifies updating and modification because it is located in one place. Inheritance makes it possible to infer all the class members not explicitly given there by searching for them in the higher classes (ancestors) in order from the most refined to the most general. This inference mechanism comes back to an algorithm for browsing the class graph according to a defined strategy.

a)Simple inheritance

Inheritance has long been seen as an inheritance of structure first and behaviour second. This is no longer the case with Java and UML, which distinguish two forms of inheritance: class inheritance is an inheritance of structures and behaviours, interface inheritance is only an inheritance of behaviours. An inherited class is generally an abstract class, which will have no instance, but which constitutes an algebraic type (a structure with operations). You can have as many levels of inheritance as you want. When a class inherits from a more abstract class, it inherits its attributes and its operations or methods.

b)Multiple inheritances

Multiple inheritance is an extension to the simple inheritance model where one class is allowed to have several parent classes to model multiple generalizations. An object can be considered from several points of view. This is the main reason we have to consider multiple inheritances. For example, the cathedral of Notre-Dame de Paris is both a work of art and a place of worship. Care must be taken to avoid homonymy, which should not mix two structures instead of giving them two different names. At first glance, it seems that one class can inherit from several classes because an object can have several parts, and the object has been attributed to the properties of its parts (metonymy). However, only the question of points of view corresponds to inheritance because if an object is composed of several parts, it will be constructed by a compositional mechanism. At the design stage, it is legitimate to describe a class inheriting from several classes. If the programming language used does not allow multiple inheritances, the problem will have to be solved at the implementation stage.

The use of multiple inheritances 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. In programming, managing multiple inheritances of structures is difficult because if inheritance causes a conflict over attributes, you have to rename an attribute in one of the classes or see the design error that causes the Conflict. If inheritance causes a conflict of methods, a conflict resolution strategy, i.e., a choice or combination procedure as in CLOS [6] [22] [43], should be used. This is why some languages such as Smalltalk or Java prohibit multiple inheritances of structures. Some languages prefix the name of the attribute by its class of origin. If multiple inheritances are allowed, it is not advisable to do multiple inheritances on several levels.

It is better to do it only for instantiable classes and that these classes are not inherited. The notion of an interface in Java avoids multiple inheritances for classes while allowing the inheritance of behaviours. An interface only defines static constants and declares abstract methods. It represents a promise of services. There can be multiple inheritances between interfaces, and a class can implement several interfaces without conflicts since no instance variable and no method is defined.

We will come back to this dimension to describe the conceptual choices of integration of multiple inheritances made for the design of the OO-Prolog language and the strategies for resolving inheritance conflicts.

C. Polymorphism

Polymorphism is that several functions can have the same name if they do the same thing on different objects. The function is then said to be generic. The form in which a function is called does not completely determine the function that will be executed since functions are generic: they only define a contract on how they behave. Their call parameters have a type that will select the concrete function that will be executed. And therefore, the same function call can trigger different methods depending on the objects passed to it. Even if the variables have a type, several objects can correspond to this type because of inheritance between classes and between classes and interfaces. The object will execute the method defined in the most specialized class of which it is a part. A generic function call must first resolve which method applies and then apply it to the call's arguments. In some cases, the decision may be made statically, once and for all, and the method call at compile-time may replace the function call. In other cases, the same call may correspond to objects of different types, and resolution can only be made at runtime.

D. The composition

When an object is composed of several parts, its composition is constructed because variables will reference the object's attributes and parts. The object's behaviour can be distributed on its parts and accessible by calling methods on the object's parts via its variables.

III. INHERITANCE SEMANTICS

Almost all object languages implement a notion of inheritance between classes. As we have just seen, the Principle is to specialize and factorize. This allows knowledge to be shared efficiently to obtain, on the one hand, a more compact code and, on the other hand, a finer representation of the problem to be solved. The programming of an application in these languages will consist of grouping the most general information into classes, which are then specialized step by step into sub-classes implementing more specific behaviours. The classes are organized in an inheritance graph, which allows visualizing the links between them. However, inheritance is a refinement mechanism whose mode of application leaves a certain number of design choices. In particular, the mode of composition of the properties must be defined. To do this, we are faced with two design choices: the semantics of inheritance [11] [12] and the path strategy of the inheritance graph, i.e., the order in which the classes will be considered.

In this section, we come back to this concept of inheritance to describe its semantics and the choices that were retained for the OO-Prolog language conception.

The traditional definition of inheritance presupposes non-monotonous semantics in the composition of the different inherited classes. This means that when a subclass redefines a method, this redefinition replaces or hides the definition already given in the overclass. Thus, if an instance of this class receives a message that must be answered by executing this method, the subclass's definition will be executed. In practice, a mechanism is often provided to override this. For example, this sends a message to super in Smalltalk-80, which explicitly designates the definition in the classes above.

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 common object languages, such as the linearization of the inheritance graph classes. Examples are ObjVProlog [48] [49] [50] and Prolog++ [66] [47]. Others support multiple inheritances and offer no means of resolving conflicts (e.g., the systems of Kowalski [44] [45] and Zaniolo [94]).

Gallery [32], Leonardi, and Mello [46] propose object-oriented logic programming to replace non-monotonous semantics with monotonous semantics backtracking would explore all the definitions vertically, from the subclasses to the superclasses. This approach is interesting from the point of view of first-order logic, which is monotonous. However, it poses a major problem. Indeed, if the inheritance is used to build based on 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 [33] to propose a new form of cut to prevent 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
- It is the most common in object languages and is necessary in many cases to differentiate objects' behaviour.

IV. TECHNIQUES FOR RESOLVING INHERITANCE CONFLICTS

Inheritance is a mechanism for both hierarchical and deductive information sharing, defined on a set of objects partially ordered by a specialization relationship. This deductive aspect is of particular interest here. Each of these classes has properties (attributes or methods): the inheritance object: the subclasses inherit them from their superclasses. As a first approximation, these properties have values (scattered in the inheritance graph) and a name (or selector).

Multiple inheritances allow more flexible modelling 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 Fig. 1 taken from [69] [25] [26], where and are two direct superclasses, both of which have the property P, each without Conflict.

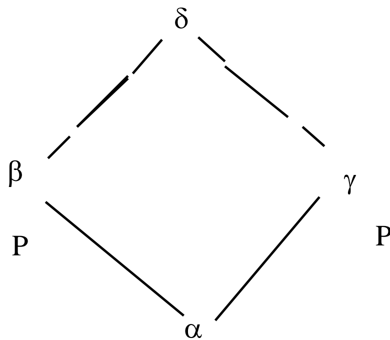

Fig. 1 Primitive scene

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 therefore generally prohibited because they are incompatible with a programming framework based on rigour and reliability. In artificial intelligence, multiple inheritances are a natural and indispensable principle for modelling real-world situations and entities. We describe below the common techniques [8][7] [59] [60] [61] [62].

A. Conflict resolution by mistake

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

B. 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.

C. Conflict resolution by renaming

Conflict resolution by renaming occurs when the language's semantics consider the same name introduced by different classes as referring to distinct fields, thus duplicating the renamed components. The expressions "conflict resolution by duplication" and "conflict resolution by renaming" are synonymous. The Eiffel language uses this Principle. The program example below shows how this is done in the Eiffel language (renaming of conflicting attributes and methods) [8].

For example :

```
CLASS Problem
        EXPORT origin, priority...
        FEATURES ...
END
CLASS Document
        EXPORT origin, priority...
        FEATURES ...
END
CLASS Of_delay
        EXPORT ...
        INHERIT
                problem RENAME origin
                AS hazard_manufacturing,
                AS priority priority1 ;
                document RENAME origin
                AS programme_fabrication,
                AS priority priority2 ;
        FEATURES .
END
```

D. Conflict resolution by qualification

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

E. 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, multiple inheritance conflicts over the Duration and Priority attributes are handled by viewpoints in

OBJLOG [27] [28] [15].

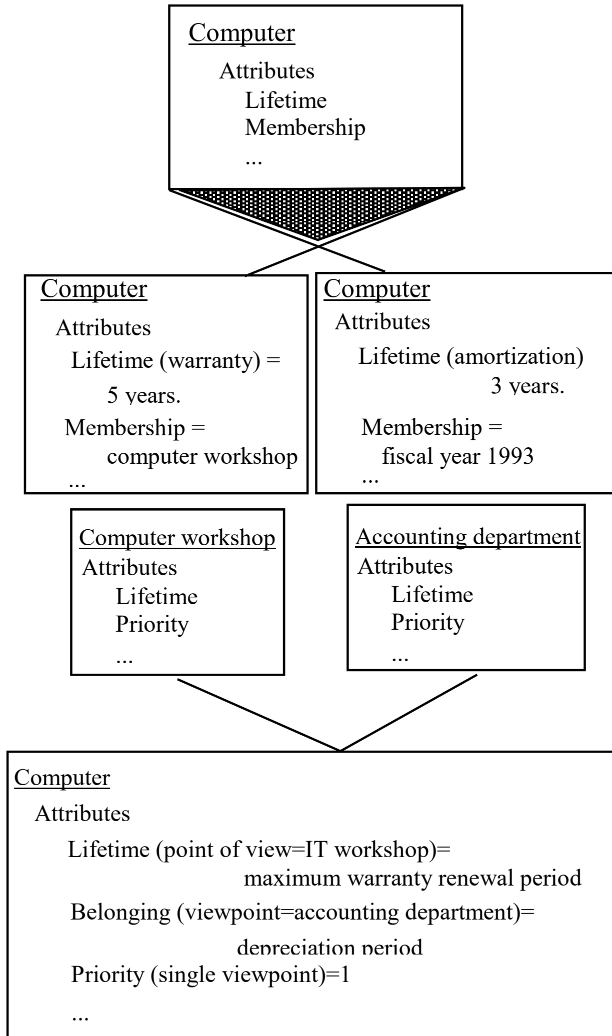

Fig. 2 Points of view

Let us imagine the Computer class (see Fig. 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 a 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. "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" [8]. A class may, therefore have several points of view. The sum of these points of view, i.e., the whole class, will be called perspective. "A perspective is a composite class representing different interpretations (points of view) of the same abstraction of the real world" [8].

Languages that resolve multiple inheritance conflicts based on their classes' decomposition into viewpoints will somehow shorten the path's qualification more intuitively than languages. All references to the selector fully qualify the source of its statement (see conflict resolution by qualification). 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 knowledge representation [84], where different types of knowledge do not have the same meaning in different domains of discourse. For example, the OBJLOG language defines a mother class as a perspective for a daughter class. Unlike CLOS, which resolves possible conflicts using a precedence list, OBJLOG enshrines the point of view. The conflict resolution algorithm will reason by difference or equivalence of points of view.

F. 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 to determine a certain sequence. It is the notion of a demon that is used here. In the KEE language, these labels aim at managing specialization to avoid arbitrary masking of the method's code (overloading) or, more generally, conflicts in multiple inheritances [8]. In this case, a parameterization of the path of the inherited classes is given by the combination. This Principle of method combination is based on the generic functions introduced in the CLOS language [6] [22] [43]. We speak of conflict resolution by method combination when the semantics of the language use method labelling (daemon) to allow certain chaining. Moreover, the combination provides a parameterization of the path of the inherited classes.

G. 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 used to respond to a message. Classically, this is equivalent to defining a total or partial order in the inheritance graph or in the subgraph 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 path algorithm's execution will be selected; hence the importance of knowing the algorithm used during programming 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 classes' priorities. Linear techniques have the major disadvantage of systematizing each Conflict's treatment without considering the semantics of the properties involved. As Masini [Masini & al. 89] points out, conflict

resolution can only be reliable if it considers the knowledge related to the application. Systematically applying a default solution cannot, therefore, correctly resolve each case. Therefore, the algorithms used in the graph must be taken into account according to the problems' nature to be solved [8]. Certain modes of conflict resolution (collisions and repeated inheritances) prevent this arbitrary choice, dictated by class specialization's chronology.

V. INHERITANCE MECHANISMS IN OO-PROLOG

OO-Prolog is one of the many hybrid languages resulting from work on the integration of object-oriented programming paradigms and logic programming paradigms [9] [39] [42] [40] [5][18] [19] [34] [38] [80] [1][2][3][31] [20] [41] [51] [56] [92] [67] [48] [49] [50] [66] [72] [73] [74] [75] [76] [77] [78] [79] [35][36][85][86][87].

OO-Prolog supporte l'héritage multiple avec une sémantique non-monotone. Pour résoudre les conflits d'héritage en OO-Prolog, nous adoptons une solution basée sur la résolution non-déterministe, sur la notion de point de vue et sur le concept de nom complet d’attribut.

Pour beaucoup de langages à objets usuels, une stratégie par défaut de parcours du graphe est nécessaire. Les stratégies linéaires restent, pour l'instant du moins, le meilleur compromis [Masini & al. 89]. Pour certains, elles sont actuellement les seules techniques acceptables [69][25][26]. Cependant, trois raisons nous amènent à proposer, pour la programmation logique par objets, une approche non-linéaire, non-déterministe. Premièrement, comme le souligne Masini, il n'existe sans doute pas une stratégie linéaire universelle, idéale, satisfaisante dans tous les cas [Masini & al. 89]. Deuxièmement, les techniques linéaires ont l'inconvénient majeur de systématiser le traitement de chaque conflit, sans tenir compte de la sémantique des données qui y sont impliquées. Enfin, la possibilité qu’offre Prolog d'explorer, par retour arrière, toutes les alternatives possibles, permet, en cas d'ambiguïtés, de considérer un objet avec tous ses points de vue (sans aucune discrimination).

OO-Prolog adopts a dynamic inheritance for both attributes and methods. However, attribute inheritance and method inheritance are treated differently.

A. Attribute inheritance

For the choice of the inheritance model of the OBJLOG language, Dugerdil and Chouraki hypothesized that the conflicting attributes do not have the same semantics [27] [28] [15]. We take up some of OBJLOG's ideas and retain this hypothesis 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 [29]. 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' (Fig. 3).

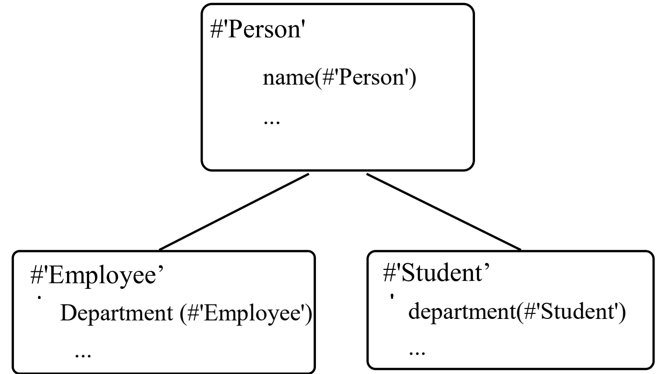

Fig. 3 Full name of an attribute in OO-Prolog

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).

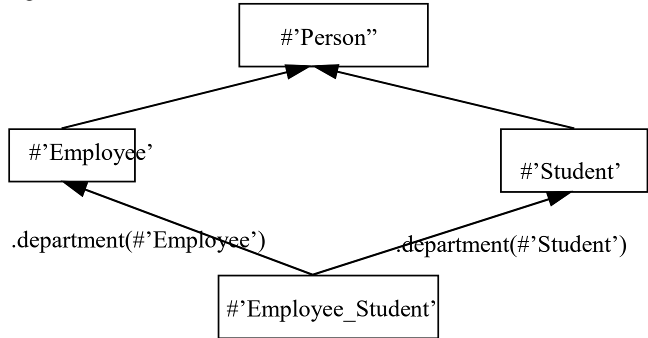

Fig. 4 Interpretation of an attribute

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 class C, i.e., the set of interpretations of the same name's attributes (but not masked) 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. In the context of Fig. 4, the disjunctive interpretation of the 'department' attribute at class level #' Employee_Student' is {#' Employee,' #' Student'}. The disjunctive interpretation of an attribute at its

definition class is the singleton composed of this same class. For example, the department attribute's disjunctive interpretation at the class level #' Employee' is the singleton {#'Employee'}. Thus, when a method called the interpretation of an attribute is a free variable, it is unified with each of the disjunctive interpretation elements at the current class level. Therefore, let's be an instance of the class #' Employee_Student, 'having for study department "La Seine-Maritime" and 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 attribute's value corresponding to each interpretation.

We then obtain:

O <- getval(department(Int),Val).

(1) {Int = # 'Employee', Val = La Haute-Seine}

(2) {Int = # 'Student',Val = La Seine-Maritime}

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

O <- getval(department(#'Employee_Student'),Val).

(1) {Val = La Haute-Seine}

(2) {Val = The Seine-Maritime}

B. The inheritance of methods

Here 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 inheritances. Behaviour inheritance is a synthesis of the consequences of the inheritance relation at the level of methods; it describes the evolution of t classes' behaviour through user-defined inheritance links [Royer 91b]. In OO-Prolog, method inheritance is also dynamic but managed differently by three complementary strategies, which can be combined dynamically.

The non-deterministic strategy

OO-Prolog uses a partial order with backtracking to consider an object with all its points of view in the case of remaining ambiguities. By default, sending a message activates all methods in Conflict, taking advantage of the Prolog interpreter's backtracking in his exhaustive search for solutions to a query. For example, in fig. 5 below, #'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?

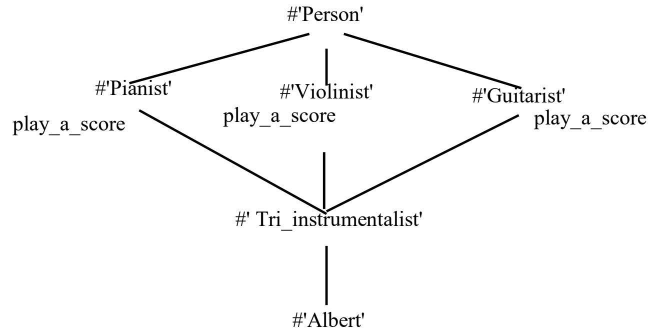

Fig. 5 Points of view of #' Albert.'

In a linear approach in which classes are given priority, Albert will consider the class with the highest priority and use the instrument corresponding to that class by default. For example, in CLOS, it will be a 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'}):

#'Albert' <- (#'Pianist'):play_a score.

or #'Albert' <- (#'Violinist'):play_a score.

or #'Albert' <- (#'Guitarist'):play_a score.

This prevents an arbitrary choice dictated by class specialization's chronology and 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 Fig. 6, sending the message department(D) to the object #' Paul' is equivalent to :

#'Paul' <- department(D) (as #'Employee')

or

#'Paul' <- department(D) (as #'Student')

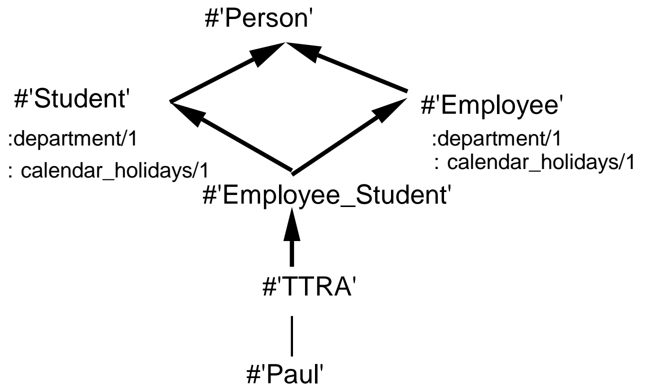

Fig. 6 Student and Employee: which department/1 instance O uses at the TTRA level?

Thus, by default, OO-Prolog does not deal with method inheritance conflicts. Sending a message activates all the conflicting methods, taking advantage of the Prolog interpreter's feedback in his exhaustive search for solutions to a query.

Thus, while in the monotonous approach, backtracking is used to introduce monotonous inheritance semantics (Fig. 7.a), 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 (Fig. 7.b).

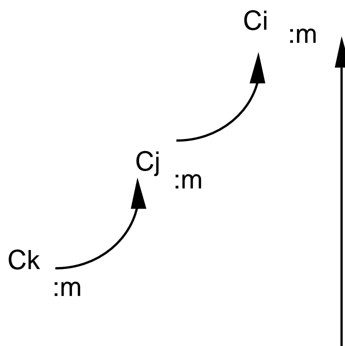

Fig. 7.a Vertical backtracking

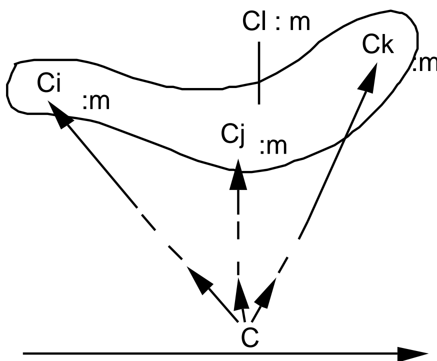

Fig. 7.b Horizontal backtracking

By default, the general rule is that sending a message triggers all possible methods, taking advantage of the Prolog interpreter's feedback in his exhaustive search for solutions to a query. For example, in the context of Fig. 6, sending the message department(D) to the object #' Paul' of the TTRA class is equivalent to or logical:

#'Paul' <- department(D) (O as Employee)

or

#'Paul' <- department(D) (O as a Student)

and is dealt with by exploring conflicting classes by backtracking. In our opinion, this strategy is more general than a classical non-monotonous linear strategy such as Pclos, P1, etc. Any solution obtained using such a linear strategy can also be a solution to this approach. For example, in the context of Fig. 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.

Linear strategy

The form "O <-- Message" is processed using a predefined linear extension algorithm. As we have already pointed out, linear strategies must be taken into account according to the problems' nature. They do not always give the same result. Therefore, the user must be given the possibility to introduce his strategies or use several existing strategies (Pclos, P1, Pflavors, etc.). The solution currently adopted in OO-Prolog consists of 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.

O <-- department(D).

{Val = La Haute-Seine}

A simplified version of the inversion algorithm consists of removing the nodes from the graph to 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 it 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.

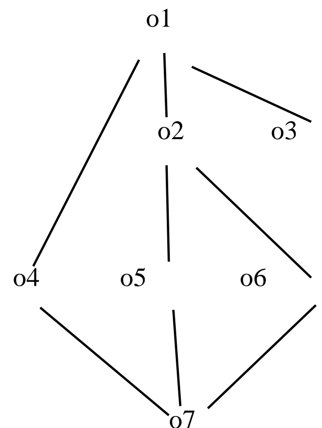

Fig. 8 Example of an inheritance graph

Let us consider the graph in Fig. 8 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 the 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. For example:

lookup(Class,Precedence,pclos) :-

% definition of the CLOS strategy.

lookup(Class,Precedence,inversion) :-

% definition of CLOS strategy.

% definition of the strategy by inversion or P1.

Thus, it is possible to define several independent linear strategies and use them in the same application. The choice of a strategy is made by assigning an environment variable the name of this strategy. The primitive set_lookup then dynamically sets the strategy to be used: set_lookup(S), S being the set strategy. For example, if the user defines CLOS's strategy, 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:

get_lookup(X),set_lookup(pclos),get_lookup(Y).

{X = inversion, Y = pclos}

true

set_lookup(pclos),get_lookup(pclos).

{}

true

This assignment is temporary and defeated by backtracking. Currently, only two linear strategies are integrated into OO-Prolog. The in-depth course with a reversal that we have described above. Other strategies, such as PCLOS, will soon be available.

The explicit designation

It consists of 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 possible to make certain choices "by hand," thanks to the other classes' horizontal masking. A designation may be incomplete. This is 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 Fig. 6, the application

O <- (#'Employee'):department(D).

{D = La Haute-Seine}

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 designations

In OO-Prolog, the explicit designation can be multiple, i.e., and several classes can be designated as follows:

<Object > <- ([class1 >, ..., Classen >]):<message >

The following examples give an illustration of this mechanism.

(1) Using the example in Fig. 5, we can write :

#'Albert' <- ([#'Pianist',#'Guitarist']):play_a score.

(2) In the context of Fig. 9 below, we can write:

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

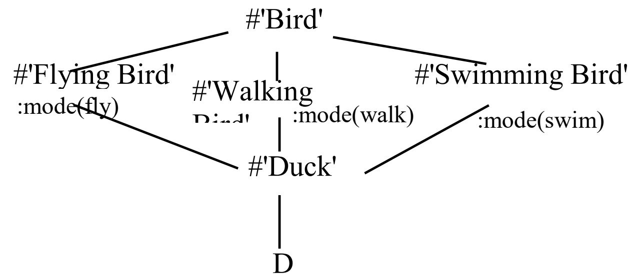

Fig. 9 Modelling the different points of view of the duck

Although the designated classes are considered in this order, it is not of great importance since the result is the same regardless of the order given. Thus, we can also write :

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

which leads to the same result, the only difference being the order in which the solutions will be rendered: {fly,swim} in the first case and {swim,fly} in the second.

(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, i.e., that the method sought is not defined in one of the designated class subclasses.

(c) Designation and path of the inheritance graph

It is also a means of reducing the complexity of the inheritance graph methods. It consists of making a jump to the designated class and reducing the method search graph, thus avoiding unnecessary visits to all the intermediate classes.

VI. APPLICATION TO THE CLASSIFICATION OF OBJECTS

OO-Prolog In this section, we describe an application to the inheritance management model we have just presented, a dynamic classification mechanism based on the multiple

specializations of objects.

Before proposing our classification scheme, we begin by defining the concepts and briefly describing two mechanisms on which we have drawn inspiration.

A. Knowledge

Knowledge can be seen as a "way of understanding and perceiving, the fact of understanding, knowing the properties, characteristics, specific features of something" the world, and a theory of knowledge as "the explanation of the relationship between thought and the outside world" [definition of the "Petit Larousse" ed. 1991, 2020].

Newell defines the knowledge of an intelligent agent in terms of the goals to be achieved:

Knowledge is] whatever can be ascribed to an agent, such that its behaviour can be computed according to the Principle of rationality. [...] Principle of rationality: if an agent knows that one of its actions will lead to one of its goals, then the agent will select that action" [71]: Knowledge is] all that can be attributed to an agent so that the agent's behaviour can be calculated according to the Principle of rationality. ...] Principle of rationality: if an agent knows that one of his actions can lead him to one of his objectives, then he chooses that action.

Knowledge can, therefore, be defined as the perception and understanding that an intelligent agent has an external world; this knowledge will enable him to behave rationally and goal-oriented.

Knowledge can, therefore, be defined as the perception and understanding that an intelligent agent has an external world; this knowledge will enable him to behave rationally and goal-oriented.

B. Representation of Knowledge

The organization of knowledge [84] into categories of similar individuals is a natural activity. Since the birth of artificial intelligence, several knowledge representation techniques and associated reasoning mechanisms have been developed. These include classical logic systems that reason by monotonous logical inferences, rule-based production systems that use forward and backward chaining mechanisms to simulate cause-effect reasoning, schemas or semantic networks, and conceptual graphs to represent knowledge by bipartite graphs. These schema systems follow analogical reasoning, and finally terminological logics and object-centered representations that use classification [17] [14] [30] [52] [53] [54] [55] [57] [NEBEL 90][QUI93] [82] as the basic reasoning mechanism.

There are specially adapted reasoning mechanisms for each of these types of representation: modus ponens deduction for logic, hypothetico-deductive reasoning for production rules, an analogy for representations by prototypes, and so on.

In declarative systems, also called "knowledge-based systems," knowledge is separated from control. This separation facilitates the modification of knowledge and the addition of new information to the base. The reasoning is entirely data-driven; it uses inference mechanisms that allow problems for no explicit procedures in the program.

A knowledge-based system is, therefore, a system with a knowledge representation part and a reasoning part.

To solve a problem, an agent reasons on an abstraction of knowledge related to the world and the situation of the problem. To represent this knowledge, the agent develops a model of the world's elements, relationships, and behaviour laws.

The representation of knowledge is, therefore, the modelling of the different elements of the real world and the determination of interpretation procedures linking the world and the model, both at the time of knowledge acquisition and model elaboration and during the manipulation of the representation (to give explanations) and, finally, when applying the results of the model to the real world. Based on this representation and an appropriate reasoning capacity, the system must adapt and exploit its environment [BRA90][BAR&81].

The knowledge represented can be of different types: concept, fact, method, model, heuristic, event, prototype, object, etc. [BRA90] [BAR&81]. It can have different modalities: static or evolving, fixed or adjustable, certain or uncertain, valid or outdated. Moreover, it can be objective or subjective.

C. Classification

The classification [17] [14] [30] [52] [53] [54] [55] [57] [NEBEL 90][QUI93] is the fundamental process of organising information during analysis. It constitutes a process of abstraction that makes it possible to group objects with the same properties and determine the range of these properties' values. These properties are of several orders:

- attributes, which have a fixed value for an object in the Class (colour, dimensions, etc.) or a variable value over time (position, speed),
- the relationships between the objects created by the classes (composition, association),
- the states that will vary over time according to the events that affect the object, and which define the possible operations on this object when it is in this state. The notion of state is not distinct from the notion of an attribute in object languages.
- Possible operations on objects.

Classification is the main reasoning mechanism for object-based representations. Classification is a process that, starting from a structured knowledge base and a new object, finds the object's appropriate location in the base.

The classification of an instance consists of finding the

most specialized classes to which the instance belongs. Classifying an instance consists of finding the classes for which it satisfies the constraints. Given a particular individual in the universe of discourse and a class structure, classification here consists of finding the most specialized classes for which the instance satisfies the constraints. This mechanism starts from an instance of which one has total or partial knowledge and a class graph. Its intention or structure describes a class. It represents a potential set of instances (those that satisfy the class structure); the class graph is induced by an order relation that must be coherent with the set inclusion relation between the different classes.

The term classification has been used to refer to three types of mechanisms:

- categorization, i.e., the grouping of objects into classes,
- the classification of classes or the insertion of a new category or class in a class graph,
- Finally, the classification of instances, which consists of finding the most appropriate membership class in the class graph, for instance.

Our work concerns this third type of classification.

Classification is one of the most powerful human reasoning activities and a fundamental mechanism of inference [16]. This mechanism is specially adapted to object-based representations. Indeed, the structuring of knowledge into classes, subclasses, and instances favours classification to recover implicit knowledge, relations between a new situation and already known situations.

D. Classification reasoning

Classificatory reasoning [Naples 92] consists of comparing new knowledge with a set of known knowledge to deduce information related to this new knowledge. Classificatory reasoning is an essential inference mechanism. Faced with a new situation, a person takes advantage of past experiences to choose actions to be taken. He determines the most appropriate position for this new situation in the structure where he memorizes those already known. Then he infers knowledge induced by this localization.

This type of reasoning is very often used in problem-solving: Knowledge of the domain is expressed by a taxonomy of known problem types, a taxonomy of solution types, and heuristic links between them. To solve a problem, a person classifies it in the taxonomy of problems, then associates it with the most appropriate solution in the taxonomy of solutions employing a heuristic, and finally refines the solution by classification.

Among the various knowledge representation techniques, object-based knowledge representations (OBKRs) offer the necessary elements for taxonomic representation: they structure knowledge of the world around two types of objects: classes and instances. Classes represent categories of similar objects and are organized by a specialization relationship within a taxonomy. Instances describe individuals, members of classes. Classificatory reasoning finds its "natural space" in object representations. Thus, the main reasoning mechanism of such a representation is the classification of instances [93]. To classify an instance consists of finding its most specialized membership classes in taxonomy and then infer knowledge related to this localization.

Our work takes place within the framework of classification reasoning and concerns the classification of instances in a representation of object-oriented knowledge. Our contribution lies in the object-based representation as well as in the mechanism of instance classification.

At the representation level, we will deal with the inheritance conflict due to multiple specializations present in most knowledge object representation systems.

When a class has several superclasses in taxonomy, and these have a common attribute, the system has to decide which superclass inherits the attribute. We argue that the source of this Conflict is the combination, in a single class graph, of several class graphs corresponding to different considerations of the same object from different points of view.

E. The classification model of TROPES

TROPES [Marino 89, 90, 93; Gensel 92, 93, 95] [54][55] [13] is an object-based representation system [68] [93] designed to support classificatory reasoning similar to that exploited by SHIRKA [Rechenmann 88; Haton 91], its predecessor, and by terminological languages [McGregor 92] [10] [83]. TROPES is based on a class/instance approach. In this model, a knowledge base is partitioned into independent concepts that model different families of individuals. Several points of view can be associated with a concept, each of them allowing for a particular interpretation.

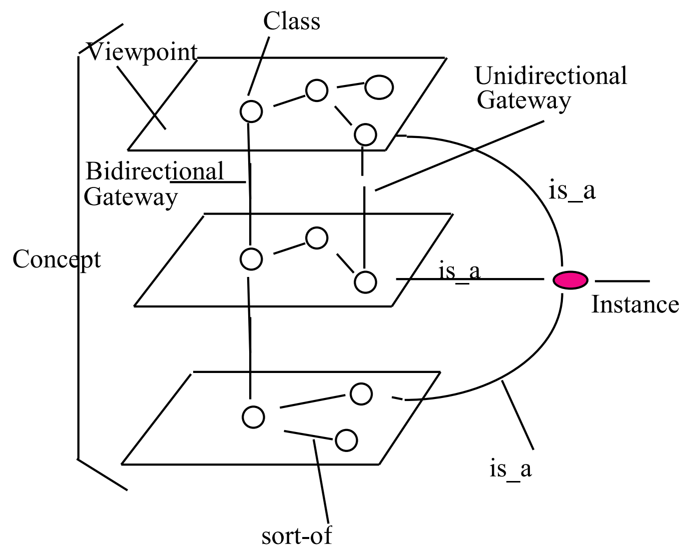

Fig. 10 The main entities of the TROPES system

A point of view corresponds to a set of classes organized in a hierarchy by the specialization relation, which translates the inclusion of all the instances of a class - the class's extension - in that of a superclass. In such a hierarchy, except for the root, a class has a single direct superclass (the hierarchy is, therefore, a tree). Moreover, extensions of direct subclasses of the same class are disjoint (multi-instantiation is impossible from the same point of view). Therefore, an instance is attached to a single class in each point of view by the "is-one" link and belongs to each class located on the latter's path at the root (Fig. 10).

Since views on a concept represent different categorizations (hierarchical class organizations) of the same set of instances, it may be desirable to emphasize the relationships between classes of distinct viewpoints. This is achieved by defining gateways. In its simplest form, a gateway requires the inclusion of a source class in a destination class.

The classification of instances into TROPES [54][55] [13] is a mechanism that consists of confronting the knowledge acquired in the instance with the different hierarchies (points of view) of classes of the concept. Classification is based on matching, which compares the attributes of the instance to be classified and the constraints imposed in the class's definition by the facets of the attributes. Concerning the belonging of an instance to a class, a class is labelled as safe if the instance checks all the constraints of the class, impossible if it does not check them, possible if the information available in the instance is not contradictory with the constraints, but is not sufficient to decide on the belonging to the class. Once the matching is done, the classification proceeds to a label propagation to save future matches. Label propagation follows the following rules:

- any upgrade from a safe class is safe,
- any subclass of an impossible class is impossible,
- any class at the same level as a safety class is impossible.

The Principle of the algorithm is therefore based on a marking of the classes as "Possible," "Impossible," or "Safe," according to the confrontation of the content of the instance with the description of the candidate class, during a descending course in each of the points of view (for more details see [Marino 89, 90, 93; Gensel 93; 81]). The descent of the instance to the class's subclasses to which it belongs leads to a permanent interaction with the user in TROPES to ask for the values for the different attributes [81]. As the classification algorithm brings the object I down to the different points of view, I's knowledge is enriched. Once new information is obtained ("Obtaining Information" phase), the matching procedure ("Matching" phase) compares the values already given by the user for the different attributes with the constraints of the attributes defined in the subclasses. For the subclasses' attributes, the system asks the user for the attributes they need. As a result of this comparison, the procedure marks each of the revised classes as "impossible" (instance I do not satisfy the class attribute constraints) or "safe" (instance I satisfy all the attribute constraints of the classes it belongs to) or they remain "possible." There is only one class that can become "safe" from each point of view. Once the classes have been marked, the procedure for updating the information of the "safe" class from each point of view retrieves all the aggregated information. Finally, the "choice of viewpoint" procedure calculates the viewpoint to be taken as current to continue the instance's classification. To carry out this choice, the strategy that has been retained in TROPES consists of ordering the points of view according to the user's knowledge of them. This is a deterministic choice. One of the reasons for our approach is to make this choice non-deterministic so that all the points of view of an object can be considered.

F. MIC (Multi-Instantiation by Classification)

MIC (Multi-Instantiation by Classification) [Rieu 91a, 91b; Olga Marino 24] [63] [64] [65] is another classification mechanism implemented in the knowledge representation system SHOOD [Escamilla 90b, 93] and which inspired our approach. It is designed for the classification of evolving objects that may be incomplete and go through incoherent states. It is based on multi-instantiation.

That of multi-instantiation replaces instantiation by perfect casting [Masini & al. 89] by flexible casting. Multi-instantiation by flexible moulding allows the classification mechanisms to be extended to simultaneously consider the notions of incomplete and incoherent objects, points of view, and knowledge evolution. Instantiation is one of the fundamental concepts of object systems [Masini & al. 89]. It allows linking an object to the conceptual entity that describes it: it is class. In most object systems, an instance is attached to only one class [37]. Multi-instantiation allows the simultaneous attachment of an object to several classes [Van De Riet 89]. We are talking here about explicit multi-instantiation, which should not be confused with implicit multi-instantiation induced by specialization links. Explicit multi-instantiation allows the attachment of an instance to classes that are not directly or indirectly specializations of each other. It thus authorizes the attachment of an instance to classes with semantics different from those drained by its creation class. It is no longer a question here of refining or questioning an object's point of view, but of enriching semantically different knowledge, i.e., taking into account a new point of view on an object. This is the case of "my aircraft," which is attached to the collectables class because it is of a respectable age. In this case, "my aircraft" can be seen as an aircraft or a collector's item (Fig. 11).

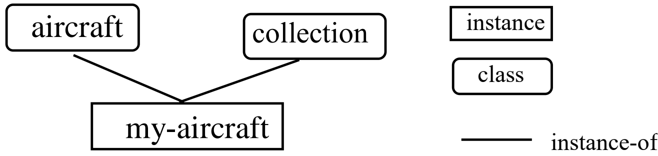

Fig. 11 Multi-instantiation by flexible moulding

Multi-instantiation was chosen as a classification mechanism in the SHOOD model, especially for :

- Preserve the identity of an object through its different points of view.
- Avoid certain hollow classes, i.e., those with a low cardinality of all instances.
- Avoid certain modifications to the class graph, particularly the classes between points of view not initially planned. If the case of collection planes has not been foreseen when designing the classes, the class of collection planes will have to be dynamically added.
- Dynamically consider new points of view as they are developed: add an instantiation link between the object and the view class's point.

However, once the instance is attached to its point of view classes, as in multiple inheritances, the problem arises from choosing a point of view when sending a message to this object. This problem is analogous to that of the management of multiple inheritances. Unless there is an explicit method, an implicit strategy must then be planned, adding processing cost. A particular reason for our approach is to avoid this double processing, especially in multiple inheritance systems. To do this, instead of multi-instantiation, we have chosen various specifications as the classification mechanism. Like multi-instantiation, numerous specifications allow the preservation of the identity of an object through its different representations. It avoids additional processing. The only problem here is the one underlined above: the dynamic addition of points of view when these are not foreseen when designing classes. However, it is possible to foresee a large number of them and avoid creating them dynamically. Moreover, since the viewpoint classes to be added dynamically are sheets of the inheritance graph, their addition does not lead to modifications of the initial graph but only create simple specialization links between this class and its superclasses.

G. DCMSO (dynamic classification by multiple) specialization of objects

OO-Prolog is equipped with a semi-automatic dynamic classification mechanism, called CDSMO (Dynamic Classification by Multiple Specialization of Objects), based on multiple specializations. This mechanism uses a scheme similar to MIC (Multi-Instantiation by Classification) [Rieu 91a].

The classification operation is a very important manipulation of a knowledge base structured in a hierarchy of classes. Classifying an instance consists of finding the classes to which it belongs (cf. fig. 12). Initially, the object I belongs to class C. Classification consists of moving the object down into C's subclasses.

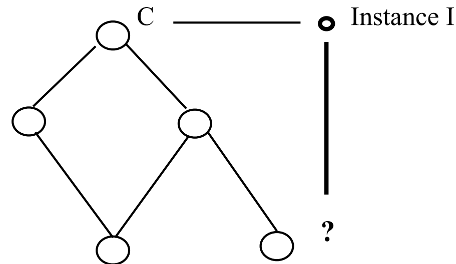

Fig. 12 Instance I already belongs to class C: The classification process seeks to move it down to the lowest subclasses of C

Depending on the type of knowledge to be added to the database, there are two classification types: classification of terminal instances and classification of classes. These two types of classification are extremely different. The classification of instances is the most important manipulation operation of a knowledge base structured in a hierarchy of classes. Classifying an instance consists of attaching it to a class hierarchy by determining its most specialized classes. Placing a class in an existing hierarchy requires modifying the links between classes, checking the database's consistency, and possibly modifying the classes that link the instances that already belong to the database. In what follows, we limit ourselves to the classification of terminal instances.

If we want to express that two classes C1 (plane) and C2 (collection) model two different points of view of the same set {i1, i2, ...} of objects, we can use :

- Multi-instantiation, by directly attaching i1, i2, ... to C1 and C2. As we have just seen, it is this solution that has been chosen in the MIC mechanism.

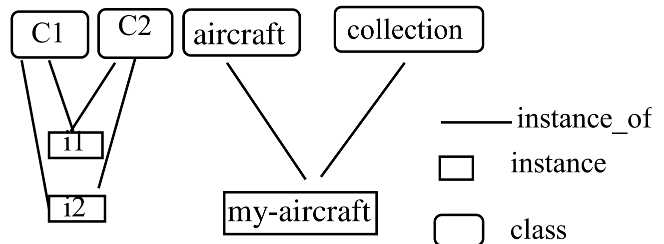

Fig. 13 Multi-point of view by multi-instance

- Aggregation, by creating a class C'12 (col-air) in which each attribute (a1 and a2) corresponds to the point of view.

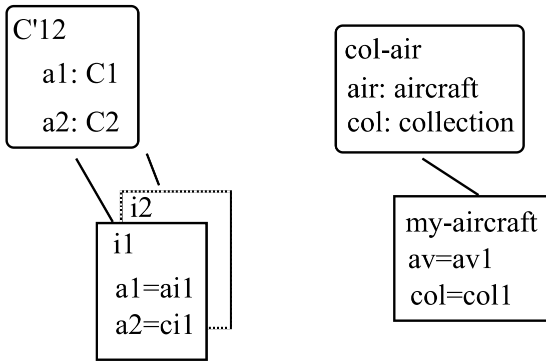

Fig. 14 Multi-point of view by aggregation

- Spécialisation multiple, par la création d'une sous-classe C12 (air-col) commune à C1 et C2 dont l'ensemble des instances est {i1, i2, ...}.

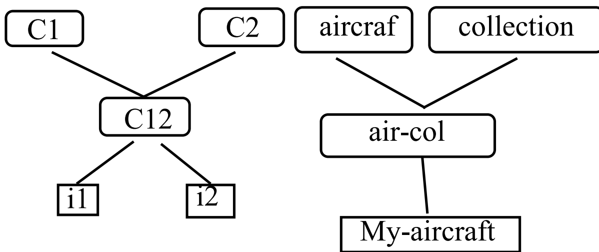

Fig. 15 Multi-specialization viewpoint

Subsequently, classes C12 (air-col) or C'12 (air-col) are called inter-view classes. Classes C1 (aircraft) or C2 (collection) are called the point of view classes. These representations induce differences, notably on the following points:

- Grouping information about different points of view in the classes. In specialization and aggregation, the different points of view cohabit within the same inter-point of the view class. This is not the case of multi-instantiation where this cohabitation is not materialized by a class but by the different instantiation links linking each instance to the viewpoint classes.
- Preservation of the identity of an object through its different points of view. In the cases of multi-instantiation and specialization, each instance i1, i2, (my-aircraft) belongs to the point of view classes C1 (aircraft) and C2 (collection): it is the same object perceived from different points of view. In aggregation, two points of view of the same object have no reason to be represented by the same identifier. Indeed, C'12 not being a subclass of C1 and C2, one of its instances has no reason to belong to C1 and C2; moreover, C1 and C2 are not in most cases specializations of each other. Aggregation, therefore, does not make it possible to express that they are indeed the same subject.

Our mechanism is based on multiple specializations. Remember that we want a mechanism that preserves the object's identity without having recourse to another inference mechanism, such as multi-instantiation, which would add processing cost. As in MIC, the execution of the classification mechanism partitions classes into:

- Impossible classes: those for which the values of the instance's attributes are in contradiction with the constraints of these attributes in the class. For a 60-year-old person, the children's class is impossible.
- Possible classes: those for which the instance is not yet in contradiction with the class.
- Safe classes: those for which the instance satisfies all the constraints of the attributes in the class.

On the other hand, a safe class can be "safe non-terminal" or "safe terminal." A class is "safe non-terminal" when its state is "safe," and it has only sub-classes with the state "safe" or "possible." A class is "terminal safe" when its status is "safe," and it is a sheet on the class graph, or when the status of all its subclasses is "impossible."

In DCMSO, the partitioning of classes into impossible classes, possible classes, and safe classes is built on the assumption that all constraints are strong, i.e., non-violable. However, when there are several candidate classes (in the sense of MIC), instead of adding to the complexity of managing multiple inheritance methods of managing multi-instantiation (two problems of almost equivalent complexity), our scheme uses multiple specializations as a classification tool. This is done by attaching the instance to be classified to an inter-point of a view class, common sub-classes of the candidate point of view classes. When such an inter-viewpoint class has not been foreseen, it is dynamically added by the system. Thanks to the specialization links between the inter-viewpoint class and the found viewpoint classes, the classified object is an instance of all its viewpoint classes.

We will distinguish two object classification cases: the case of simple objects, i.e., not composite, and composite objects.

Classification: a simple case

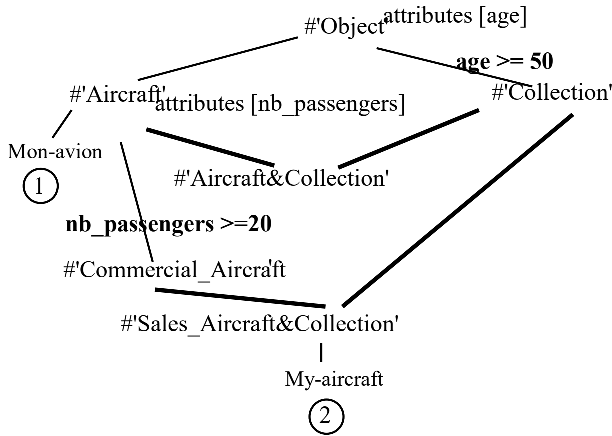

Fig. 16 representation of classification constraints and evolution of the graph when classifying objects

Let the instance to be classified. In the simple case, we consider that no object in the database depends on I (in the restricted sense), i.e., there is no terminal object I' such that one of its attributes has the value I or a list containing I. To illustrate this, we will consider the graph in Fig. 16.

In this example, the classification constraints are as follows, expressed as rules:

- an instance I of Object belongs to Collection If I.age >= 50
- an instance I of Aircraft is a Commercial_Aircraft If I.nb_passenger >= 20

?- # 'Aircraft' <- create(My_aircraft, [age(_) := 60,nb_passengers(_) := 80]),

My_aircraft <- display.

TERMINAL : : #[# 'Aircraft',1]

nb_passengers(# 'Avion') <- 80,

age(# 'Object') <- 60,

class(# 'Object') <- # 'Commercial_Aircraft & Collection

{My_aircraft = #[#'Aircraft',1]}

Unlike the new method, the instantiation create method automatically results in the classification of the created object. In the example above, the object My_aircraft, created by the class Aircraft (instantiation class), satisfies the above constraints:

- My_aircraft.age (= 60) >= 50 (My_ aircraft is instance of Object)
- My_ aircraft.nb_passengers (= 80) so >= 20).

The classes commercial_aircraft and collection are marked as safe. My_aircraft is then attached to the inter-view class #' Commercial_Aircraft&Collection' (representation class) and thus becomes both a commercial aircraft and a collector's item.

Recursive classification

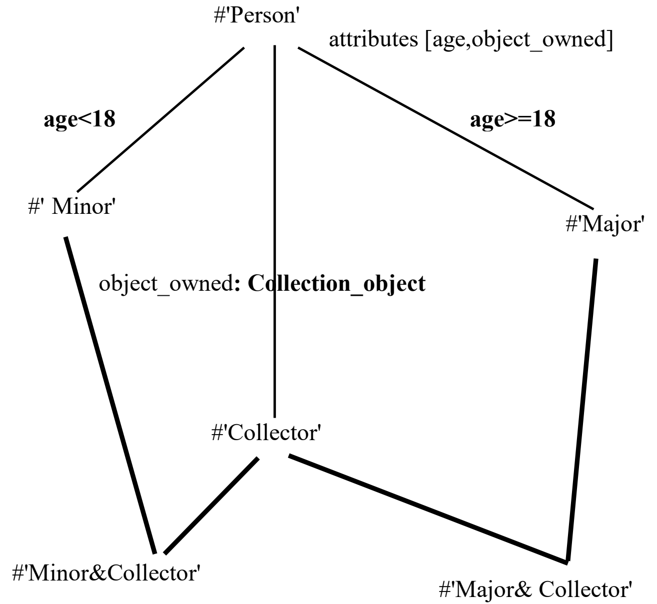

Fig. 17: Classification of composite objects

In the case where there is a terminal instance I' that depends on me, two classifications can be distinguished. The first is minimal because it only does what is necessary to position the instance to be classified in the class hierarchy (in particular, the classification of components is directed by the initial classification). The second seeks to classify equally and recursively the objects that depend on the classified instance, i.e., those that share the object that has just been classified. Indeed, if I' shares I, the classification of I may lead to modifications in me.' It is therefore necessary to reclassify the instance I' taking into account the new nature of I.

The following example is an illustration of this. For example, suppose that at least 50 years old are collection objects (belonging to the class Collection_of_objects). Then, the creation of an instance P of the class Person, with a 52-year-old plane (thus an instance of Collection_of_object) will automatically attach P to the Collector class (see Fig. 17).

?- #’Aircraft’ <- create(My_aircraft,[age(_) := 60,nb_passengers(_) := 80]),

#’Person’<- create(P,[age(_):=21,fortune(_):=200000,owns(_):=[My_aircraft]]),

P <- display,

```
My_aircraft <- display.

TERMINAL :: #[#'Person',1]
    age(#'Person') <- 21
    fortune(#'Person') <- 200000
    owns(#'Person') <- [#[#'Aircraft',2]]
    class(#'Object') <- #'Major_Rich&Collector'

TERMINAL :: #[#'Aircraft',2]
    nb_passengers(#'Aircraft') <- 80
    age(#'Object') <- 60
    class(#'Object') <- #'Commercial_Aircraft & Collection_Objet'

My_aircraft = #[#'Aircraft',2]
P = #[#'Person',1]
```

(a) Management of cycles in the classification algorithm

The designed algorithm considers possible cycles in the classification process (cases where there are mutually dependent objects). Indeed, let Oi and Oj be two mutually dependent objects (see Fig. 18).

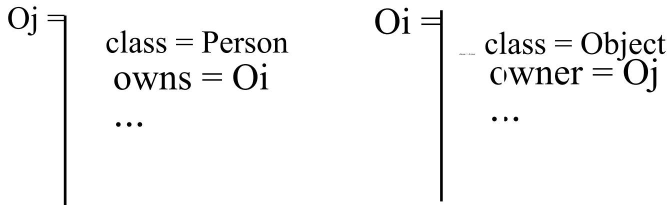

Fig. 18 Mutually dependent objects

Then the classification of Oi leads to Oj classification, and the classification of Oj has the effect of reclassifying Oi and so on. The result is an infinite loop. To correct this defect, we adopt the following solution. If Oj is classified after classifying Oi, only reclassify Oi if any classifications that took place after the classification of Oi have changed the state of Oj. This choice is justified because if after the classification of Oi and the classification of Oj (including those taking place in between) Oj has not been modified, then there is no justification for a new classification of Oi from Oi onwards. On the other hand, if the state of Oj has been changed, this may imply changes to the state of Oi and therefore requires a new classification of Oi.

Delayed classification

In the example above, the My_aircraft instance is shared by the object P. The My_aircraft object is known before the classification of P, which allows this information to be taken into account when classifying P. If we create P before My_aircraft when classifying P, My_aircraft being a free variable, the classification constraint "P has a collection object" will be delayed since My_aircraft is still unknown. In this case, P's classification is delayed for this constraint and will resume as soon as My_aircraft is known. In the end, we obtain the same result as before.

```
?- #'Person' <- new(P,[
            age(_):=21
            fortune(_):=200000
            owns(_):=[My_aircraft]])
    #'Aircraft' <- new(My_aircraft,[
            age(_) := 60,
            nb_passengers(_) := 80]),
    (P, My_aircraft) <- display.

TERMINAL :: #[#'Person',1]
    age(#'Person') <- 21
    fortune(#'Person') <- 200000
    owns(#'Person') <- [#[#'Aircraft',2]]
    class(#'Object') <- #'Major&Rich&Collector'

TERMINAL :: #[#'Aircraft',2]
    nb_passengers(#'Aircraft') <- 80
    age(#'Object') <- 60
    class(#'Object') <- #'Commercial_Aircraft & Collection_of_objects'

P = #[#'Person',1]
My_aircraft = #[#'Aircraft',2]
```

The delay mechanism allows here to freeze all the classification constraints corresponding to the My_Aircraft object.

VII. CONCLUSION

This paper has described a new model for managing multiple inheritances and its application to classification. As a general algorithmic method is probably impossible to build, an alternative is to propose more open strategies. From this point of view, our approach brings many advantages over classical or traditional methods (e.g., graph linearization). It brings a lot of flexibility to the treatment of inheritance since it does not impose a user's systematic choice. Our inheritance management method prevents an arbitrary choice dictated by

the system. It is even possible to combine different strategies depending on the problem to be treated.

The proposed classification mechanism is based on multiple specializations. Indeed, we have seen that in the DCMSO model, the partitioning of classes into impossible classes, possible classes, and safe classes is built on the assumption that all constraints are strong, i.e., non-violable. However, when there are several candidate classes (in the sense of MIC), instead of adding to the complexity of managing multiple inheritance methods of managing multi-instantiation (two problems of almost equivalent complexity), our scheme uses multiple specializations as a classification tool. This is done by attaching the instance to be classified to an inter-point of a view class, common sub-classes of the candidate point of view classes. When such an inter-viewpoint class has not been foreseen, it is dynamically added by the system. Thanks to the specialization links between the inter-viewpoint class and the found viewpoint classes, the classified object is an instance of all its viewpoint classes. Remember that we want a mechanism that preserves the object identity without having recourse to another inference mechanism, such as multi-instantiation, which would add processing cost.

The designed classification algorithm considers possible cycles in the classification process (cases with mutually dependent objects).

The delay mechanism allows all classification constraints to be frozen and triggered only when missing data are available, i.e., when the variables are instantiated.

ACKNOWLEDGMENT

The author wishes to thank Habib Abdulrab, Jean-Pierre Pécuchet, Abdenbi Drissi-Talbi, Mohamed Rezrazi, Fabrice Sebbe, and all his friends and colleagues for their help and support. He also wishes to thank Olga, Michel, Marielle, and Guyriel, who has always been very precious to realize this work.

REFERENCES

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

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

[3] H. A¨ıt-Kaci, B. Dumant, R. Meyer, A. Podelski, P. Van Roy. "The Wild LIFE Handbook," Paris Research Laboratory, prepublication edition, March 1994.

[4] V. Alexiev. "Mutable Object State for Object-Oriented Logic Programming: A Survey." Technical Report TR 93-15, Dept. of Comp. Science, Univ. of Alberta, 16 Aug 1993.

[5] J.M.Andreoli, R.Pareschi, "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.

[6] 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

[7] G. Booch "Object-Oriented Design with applications" The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1992.

[8] Bouché M., "La démarche objet. Concepts et outils.", AFNOR, 1994.

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

[10] Brachman R. J. and Schmolze J. G. "An overview of the KL-ONE Knowledge representation system," Cognitive Science, 9(2):171-216, 1985.

[11] CARDELLI L., "A semantics of Multiple Inheritance, LNCS", Vol.137, Springer-Verlag, pp.51-67, 1984.

[12] CARDELLI L., "Typechecking Dependent Types and Subtypes, in Foundations of Logic and Functional Programming Workshop", LNCS, vol. 306, Springer-Verlag, pp.44-57, 1985.

[13] CARDOSO R., MARIÑO O., QUINTERO A., Corrección y completud de la classification multi-puntos de vista de TROPES, Rapport Interne, Département d'Informatique, Université des Andes, Bogotá, 1992.

[14] 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.

[15] CHOURAQUI E., DUGERDIL Ph., Conflict solving in a Frame-like Multiple Inheritance System, ECAI, Munich, pp.226-232, 1988.

[16] CLANCEY W.J., Heuristic Classification, Artificial Intelligence Journal, Vol. 27, n°. 4, 1985.

[17] CRUYPENNINCK F., Interface de visualisation et explication du raisonnement par classification d'objets complexes, Mémoire d'Ingénieur en Informatique, Conservatoire National d'Arts et Métiers, CNAM, 1992.

[18] W. Chen and D. S. Warren. 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.

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

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

[21] L. Dekker, « 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.

[22] Linda G. DeMichiel and Richard P. Gabriel, "The Common Lisp Object System: An Overview," ECOOP, 1987.

[23] 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.

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

[25] DUCOURNAU R., HABIB M., La Multiplicité de l'héritage dans Les Langages à Objets. TSI, vol. 8, n°.1, janvier, pp. 41-62, 1989.

[26] DUCOURNAU R., Héritages et représentations, Mémoire, Diplôme d'Habilitation à diriger des recherches, spécialité : Informatique, Université Montpellier II, 1993.

[27] 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.

[28] 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.

[29] 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.

[30] EUZENAT J., Classification dans les représentations par objets : produits de systèmes classificatoires, Rapport interne, Equipe SHERPA, INRIA, 1993.

[31] A. A. Fernandes, N. W. Paton, M. H. Williams, A. Bowles. "Approaches to Deductive Object-Oriented Databases", Information and Software Technology, 34(12):787–803, 1992.

[32] Gallaire, H. "Merging Objects and Logic Programming: Relational Semantics, Performance and Standardization". In Proc. AAAI'86, pp.754-758, Philadelphia, Pensylrania, 1986.
[33] Gandilhon T. “Proposition d’une extension objet minimale pour Prolog.”, Actes du séminaire de Programmation logique, Trégastel (mai 1987), pp. 483-506.
[34] [34] Gandriau, M. "CIEL: classes et instances en logique". Thèse de Doctorat, ENSEEIHT 1988, 151p.
[35] [35] Gloess, P.Y. "Contribution à l'optimisation de mécanisme de raisonnement dans des structures spécialisées de représentation de connaissances". Thèse d'état, Univ. de TechnWorldLogie de Compiègne, Janv. 1990.
[36] [36] Gloess, P.Y. M. Oros, C.M. LI, “U-Log3 = DataLog + Constraints”, (Prototype) Actes des JFPL95, Dijon (France), pp. 369-372.
[37] Goldberg, A. and Robson, D. "Smalltalk-80: The language and its implementation". Addison-Wesley, 1983.
[38] J. Grant and T. K. Sellis. Extended database logic. Complex objects and deduction. Information Sciences, 52(1):85 110, Oct. 1990.
[39] Ishikawa, Y. et Tokoro, M. Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation, Object-Oriented Concurrent Programming (1987), W 159 198.
[40] R. Iwanaga and O. Nakazawa. Development of the object-oriented logic programming language CESP. Oki Technical Review, 58(142):39 44, Nov. 1991.
[41] R. Jungclaus. Logic-Based Modeling of Dynamic Object Systems. PhD thesis, Technical University Braunschweig, Germany, 1993.
[42] [42] K. M. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow. VULCAN: Logical concurrent objects. In B. Shriver and P. Wegner, editors, Research Directions in Object-Oriented Programming, pages 75 112, Cambridge, MA, 1987. MIT Press. (Also Chap. 30 in [86a])
[43] Sonja E. Keene, “Object-Oriented Programming in Common Lisp: a Programmer’s Guide to CLOS”, Addison-Wesley, 1989.
[44] Kowalski, R. "Algorithm = Logic + Control", Comm. ACM 22, 7 (1979), 424-436.
[45] Kowalski, R. "Logic for problem solving". North-Holland, Amsterdam, 1979.
[46] L. Leonardi and P. Mello, "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
[47] Prolog++ toolkit, an expressive and powerful object-oriented programming system, which combines the best of AI and OOPs. 47. http://www.lpa.co.uk/ppp.htm
[48] Malenfant, J. ObjVProlog-V: un modèle uniforme de métaclasses, classes et Instances adapté à la programmation logique, Université de Montréal, Dép. I.R.O., Pap. de Pech. 671 (Janvier 1989), 58 p.
[49] J. Malenfant, G. Lapalme, and J. Vaucher. OBJVPROLOG: Metaclasses in logic. In S. Cook, editor, European Conference on Object-Oriented Programming (ECOOP'89), pages 257 269, Nottingham, UK, July 1989.
[50] 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.
[51] P. Mancarella, A. Raffaetà, et F. Turini LOO: Un langage orienté objet Programmation Logique . Actes de 1995 conjointe GULP-PRODE Conférence sur la programmation déclarative (MI Sessa et M. Alpuente Frasnedo, eds), pp271-282, 1995.
[52] MARIÑO O., Classification d’objets dans un modèle multi-points de vue, Rapport de DEA d’informatique, INPG, Grenoble, 1989.
[53] MARIÑO O., RECHENMANN F., UVIETTA P. Multiple perspectives and classification mechanism in object-oriented representation, 9th ECAI, pp.425-430, Stockholm 1990.
[54] MARIÑO O., Classification d’objets composites dans un système de représentation de connaissances multi-points de vue, RFIA’91, Lyon-Villeurbanne, pp. 233-242, 1991.
[55] MASINI G., NAPOLI A. COLNET D. LEONARD D., TOMBRE K., Les langages à objets. Intereditions, Paris, 1989.
[56] F. G. McCabe. Logic & Objects. International Series in Computer Science. Prentice-Hall, 1992.
[57] MAC GREGOR R.M., BURSTEIN M.H. Using a Description Classifier to Enhance Knowledge Representation, IEEE Expert Intelligent Systems and Applications, juin, 1991.
[58] MAC GREGOR R.M., BRILL D., Recognition Algorithms for the LOOM Classifier, AAAI, San José, CA, Juillet, pp.774-779, 1992.
[59] Meyer B. “Eiffel: Programming for reusability and extendibility.”, ACM SIGPLAN Notices, 22(2):85-94, 1987.
[60] B. Meyer, “Reusability: The Case for object-oriented Design”, IEEESoftware 4, 2 (Mars 1987), pp.50 64.
[61] B. Meyer. “Object-Oriented Software Construction”. Prentice-Hall, New York, 1988.
[62] Meyer B. “Conception et programmation par objets, pour le génie logiciel de qualité”, InterEditions, Paris 1990.
[63] Alexei A. Morozov, “Actor Prolog: An object-oriented language with the classical declarative semantics”, In Sagonas K, Tarau P, eds. Proc IDL Workshop, Paris, France: 1999: 39-53. Source: http://www.cplire.ru/Lab144/paris.pdf.
[64] Alexei A. Morozov, “Actor Prolog: an object-oriented language with the classical declarative semantics”, ResearchGate, Jul 2001 (https://www.researchgate.net/scientific-contributions/28317372_Alexey_A_Morozov)
[65] Alexei A. Morozov, Olga Sushkova, “Development of Agent Logic Programming Means for Heterogeneous Multichannel Intelligent Visual Surveillance”, Proceedings of the 16th Ibero-American Conference on AI, Trujillo, Peru, November 13-16, Jan 2018.
[66] Moss C., "Prolog++: The Power of Object-Oriented and Logic Programming", Addison-Wesley, 1994.
[67] P. Moura. “Logtalk Object-oriented Programming in Prolog”. Centre for Informatics and Systems, University of Coimbra, Coimbra, Portugal, July 1999. (http://www.ci.uc.pt/logtalk/logtalk.html).
[68] A. Napoli, « 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.
[69] [69] 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.
[70] [70] NEBEL B., Reasoning and Revision in Hybrid Representation Systems, Lecture Notes in Artificial Intelligence, LNCS, vol. 422, Springer-Verlag, Berlin, 1990.
[71] NEWELL A., The Knowledge Level, Artificial Intelligence, Vol. 2 n°. 2, pp. 1-20, 1981.
[72] Ngomo M. , Pécuchet J-P. & Drissi-Talbi A. "Une approche déclarative et non-déterministe de la programmation logique par objets mutables". Actes des 4èmes Journées Francophones de Programmation Logique et Journées d’étude Programmation par contraintes et applications industrielles, Prototype JFPLC'95, pp.391-396, Dijon, 1995, France.
[73] Ngomo M. , Pécuchet J-P. & Drissi-Talbi A. "La gestion de l'héritage multiple en ObjTL". RPO’95 dans les Actes des 15èmes Journées Internationales IA’95, pp.261-270, Montpellier 1995, France.
[74] Ngomo M., Pécuchet J-P., Drissi-Talbi A. "Intégration des paradigmes de programmation logique et de programmation par objets : une approche déclarative et non-déterministe". Actes du 2ème Congrès bienal de l’Association Française des Sciences et Technologies de l’Information et des Systèmes, AFCET - Technologie Objet - 95, pp.85-94, Toulouse 1995, France.
[75] Ngomo M. "Intégration de la programmation logique et de la programmation par objets : étude, conception et implantation". Thèse de Doctorat d’Informatique, Université - INSA de Rouen, Décembre 1996.
[76] Macaire Ngomo and Habib Abdulrab, "A DECLARATIVE APPROACH OF DYNAMIC LOGIC OBJECTS", International Journal of Engineering Sciences & Research Technology (IJESRT), ISSN: 2277-9655, 7(4): April, pp.764-785, 2018, DOI: 10.5281/zenodo.1228893

[77] Macaire Ngomo and Habib Abdulrab, "A Declarative Approach of Dynamic Logic Objects", International Research Journal of 2), pp.101-115, 2018.

[78] Macaire Ngomo and Habib Abdulrab, "A full declarative approach of dynamic logic objects", International Journal of Current Research in Life Sciences (IJCRL), ISSN: 2319-9490, Vol. 07, No. 05, pp.2036-2051, May 2018.

[79] Macaire Ngomo and Habib Abdulrab, "MODELLING AND IMPLEMENTATION OF DYNAMIC LOGIC OBJECTS IN THE COMPLETE DECLARATIVE APPROACH', Global Journal of Engineering Science and Research Management (GJES), ISSN 2349-4506, pp.77-785, 25(4): April 2018, DOI: 10.5281/zenodo.1238633

[80] D. Pountain. “Adding Objects to Prolog”, Byte, 15(8), 1990.

[81] QUINTERO A., Parallélisation de la classification d’objets dans un modèle de connaissances multi-points de vue, Thèse d’informatique, Université Joseph Fourier, Grenoble, juin 1993.

[82] ROSSAZZA Jean-Paul,Utilisation de hiérarchies de classes floues pour la représentation de connaissances imprécises et sujettes à exceptions : le système SORCIER, Thèse d’Informatique, Université Paul Sabatier de Toulouse, 1990.

[83] SCHMOLZE J.G., LIPKIS T.A, Classification in the KL-ONE Knowledge Representation System, in Proceedings of the 8th. IJCAI, Karlsruhe, Germany, 1983.

[84] VOGEL C., Génie Cognitif, Collection Sciences cognitives, MASSON, PAris, pp.97 1988.

[85] Shapiro, E. "Concurrent Prolog: A progress report". IEEE Computer Advanced Engineering and Science (IRJAES), ISSN: 2455-9024, (Vol.3, No. 19, pp. 44-58, Aug. 1986. (Also Chap. 5 in

[86] E. Shapiro, (Editor), "Concurrent Prolog", Vol. 1 and 2, MIT Press, 1987.

[87] E. Shapiro, "The family of Concurrent logic programming languages", Technical Report CS89-08, Depart. of Applied Mathematics and Computer Science, The Wietzmann Institute, Rehovot, 1989.

[88] SICStus Prolog, state-of-the-art, ISO standard compliant, Prolog development system. https://sicstus.sics.se/

[89] Steele G. L. "Common Lisp: the language" second edition, Digital Press, 1990.

[90] Sterling, L. et Shapiro, E. "L'Art de Prolog". MASSON 1990.

[91] SWI-Prolog pour le web sémantic / SWI-Prolog for (sémantic) web, 2017.

[92] T. Uustalu. Combining object-oriented and logic paradigms: A modal logic programming approach. In O. L. Madsen, editor, European Conference on Object-Oriented Programming (ECOOP'92), pages 98 113, June 1992.

[93] WEGNER, P., The Object-Oriented Classification Paradigm, dans Research Directions in Object-Oriented Programming, Bruce Shiver, Peter Wegner (éd.), The MIT Press, Cambridge, MA, 1987.

[94] Zaniolo, C. "Object-Oriented Programming in Prolog". In Proc. of the IEEE International Symposium on Logic Programming, pp. 265-270, Atlantic City, New Jersey, 1984.