International Journal of Computer Trends and Technology
ISSN: 2231 – 2803 / https://doi.org/10.14445/22312803/IJCTT-V69I10P101

Volume 69 Issue 10, 1-11, Oct, 2021
© 2021 Seventh Sense Research Group®

Original Article

Multiple Inheritance Mechanisms in Logic Objects Approach Based on Multiple Specializations of Objects

Macaire Ngomo

CM IT COUNCIL – Engineering and Innovation Department – 32 rue Milford Haven 10100 Romilly sur Seine, France

Received Date: 03 September 2021
Revised Date: 04 October 2021
Accepted Date: 15 October 2021

Abstract - *This study takes place within the representation of knowledge by objects and, particularly, 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 a number of design choices. In the context of this marriage, we describe the semantics of multiple inheritances in a non-deterministic approach based on multiple specifications of logical objects. We also describe the conceptual choices for integrating multiple inheritances 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 inheritances, Multi-point of view, The semantics of multiple inheritances.*

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) [72-79] 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 [52-58, 95-98, 14, 17, 21, 24, 30, 35, 68, 70]. This representation must therefore allow a dynamic classification of logical objects and follow classificatory reasoning. Reasoning by classification consists of finding the most specialised 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, explicit naming, and 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]) or logical (example with PROLOG [88, 91, 90]), etc. However, it requires a greater capacity for abstraction than imperative or functional programming 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 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

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

80 [37], 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. The composition was later added as an autonomous dimension with UML and in modern languages such as Java.

A. Encapsulation

In the object paradigm, encapsulation concerns grouping variables and functions into classes and classes and interfaces into packages. Classes, functions, and packages are also namespaces that ensure uniqueness within the names of the elements that compose 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 to each other and ensures the protection and partial visibility of the elements outside. Encapsulation ensures the independence between the layout of a class, a function, a package, and how it is presented in relation to the other objects using it. The public presentation 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 the levels of access (private, public...) to which it gives rise facilitate the reusability of software elements and the evolution of the software.

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 behaviors introduced and facilitates 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 behavior (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) 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.

Inheritance has long been seen as an inheritance of structure first and behavior 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 behaviors, interface inheritance is only an inheritance of behaviors. An inherited class is generally an abstract class with no instance but constitutes an algebraic structure (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.

Multiple inheritances extend the simple inheritance model where one class can have several parent classes to model multiple generalizations. An object can be considered from several points of view, so we have to consider multiple inheritances. For example, the cathedral of Notre-Dame de Paris is both a work of art and 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 attributed 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. It is legitimate to describe a class that inherits from several classes; if the programming language 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, naming collisions need to be resolved if the two base classes have attributes or methods with the same name. In programming, managing multiple inheritances of structures is a difficult problem 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 inheritance 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 using multiple inheritances for classes while allowing the inheritance of behaviors. 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 or method is defined.

We will come back to this dimension to describe the conceptual choices of integrating 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 classes of objects can correspond to this type because of inheritance between classes and between classes and interfaces, and 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 the question of which method applies and then apply the method 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 done at runtime.

D. The Composition

When an object is composed of several parts, it is constructed by its parts because variables will reference not attributes of the object but parts of the object. The object's behavior can be distributed on its parts and accessible by calling methods on the parts from the object 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. Therefore, the programming of an application in these languages will consist of grouping the most general information into classes that are then specialized step by step into sub-classes implementing more specific behaviors. The classes are organized in an inheritance graph which allows us to visualize 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 conception of the OO-Prolog language.

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, for example, this redefinition replaces or hides the definition already given in the overclass. Thus, if an instance of this class receives a message which must be answered by executing this method, the definition of the subclass will be executed. In practice, a mechanism is often provided to override this. This is, for example, sending 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 those of common object languages, such as the linearization of the classes of the inheritance graph. Examples are ObjVProlog [48-50] 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]).

Gallaire [32], Leonardi, and Mello [46] propose, in object-oriented logic programming, to replace non-monotonous semantics with monotonous semantics where, by backtracking, one 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 an inheritance is used to build based on another class, which supports the idea of monotonous semantics, it is also used to differentiate behaviors. An entity is often modeled by a class, saying: my instances will be like those of such and such a class (inheritance) except for such and such behavior (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 the backtracking of 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.

IV. TECHNIQUES FOR RESOLVING INHERITANCE CONFLICTS

Inheritance is a mechanism for 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) that are 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 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 Fig. 1 taken from [25, 26, 69], where and are two direct superclasses, both of which have the property P, each without conflict.

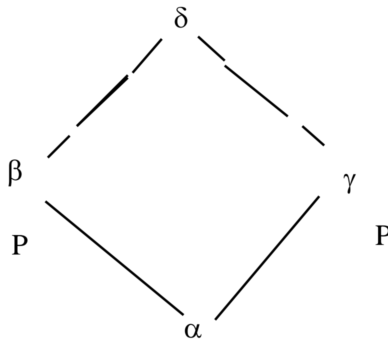

Fig. 1 Primitive scene

There is no universal technique for resolving these kinds of conflicts, and there are many 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 modeling real-world situations and entities. We describe below the common techniques [59-62, 8, 7].

A. Conflict Resolution by Mistake

Error-based conflict resolution occurs when the semantics consider the collision to be 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 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. The program example below shows how this is done in the Eiffel language (renaming 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 name of the overclass, 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 [15, 27, 28].

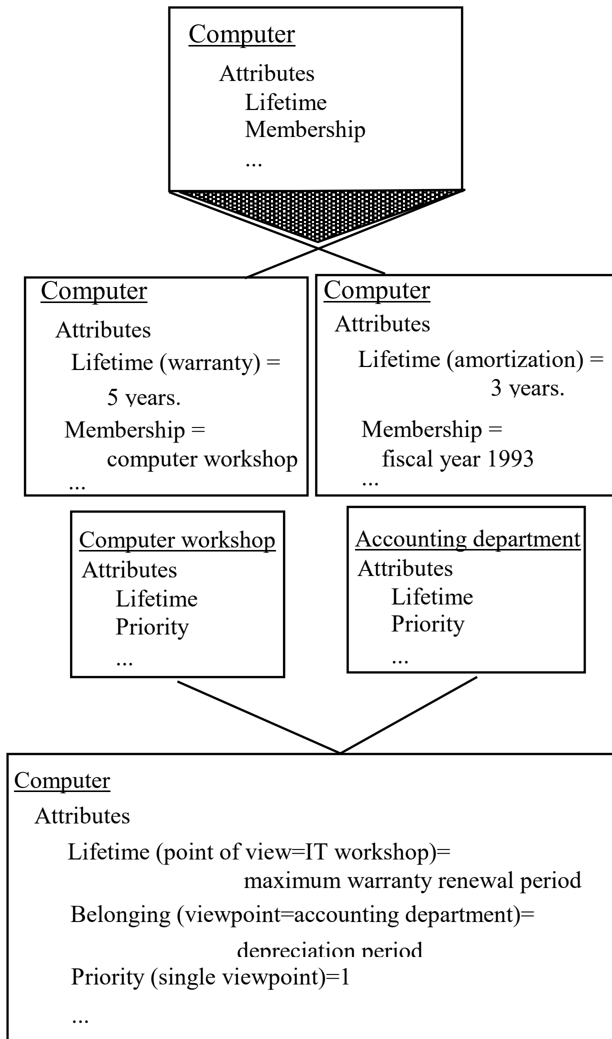

Fig. 2 Points of view

Let us imagine the Computer class (see Fig. 2), and 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. "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 the decomposition of their classes into viewpoints will somehow shorten the qualification of the path in a more intuitive way than languages where 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 modeling of perspective classes decomposed by the delimitation of points of view. This concept is fundamental in the problem of knowledge representation [84], 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 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 in the FLAVORS system, consists of labelling the methods to determine a certain sequence. It is the notion of 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 at the basis of 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 the notion of method labeling (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 execution of the path algorithm will be selected; hence is important to know 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 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 [55] 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 nature of the problems to be solved [8].

Certain modes of conflict resolution (collisions and repeated inheritances) prevent this arbitrary choice, dictated by the chronology of class specialization.

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 [1-3, 34-36, 38-42, 48-51, 72-80, 85-87, 5, 9, 18, 19, 20, 31, 56, 66, 67, 92]. OO-Prolog supports multiple inheritance with non-monotonic semantics. To resolve inheritance conflicts in OO-Prolog, 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 [55]. For some, they are currently the only acceptable techniques [69, 25, 26]. However, three reasons lead us to propose a non-linear, non-deterministic approach for object-oriented logic programming. As Masini points out, there is probably no universal, ideal linear strategy that is satisfactory in all cases [55]. Secondly, linear techniques have the major drawback of systematizing 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.

A. Attribute Inheritance

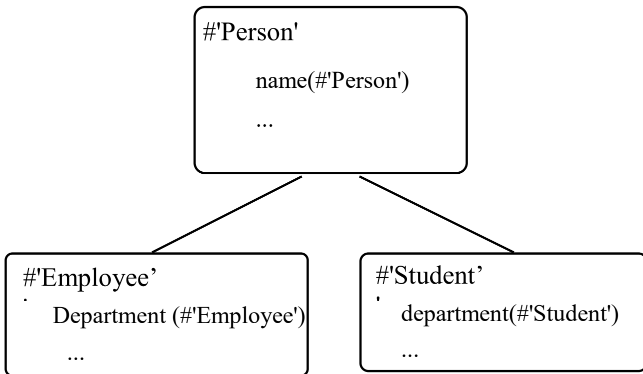

Fig. 3 Full name of an attribute in OO-Prolog

For the choice of the inheritance model of the OBJLOG language, Dugerdil, and Chourakihypothesised that the conflicting attributes do not have the same semantics [15, 27, 28]. 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 for the 'department' attributes defined in the classes #' Employee' and #' Student' (Fig. 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).

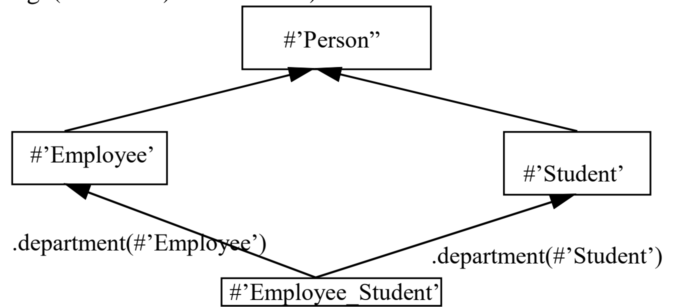

Fig. 4 Interpretation of an attribute

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 attributes of the same name (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 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 the interpretation of an attribute is a free variable in a method call, it is unified with each of the elements of the disjunctive interpretation of this attribute at the current class level. 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.

  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 level of the subclass 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 behavior. We are, in the most general case, that of multiple inheritances. Behavior inheritance is a synthesis of the consequences of the inheritance relation at the level of methods; it describes the evolution of the behavior 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.

a) The Non-Deterministic Strategy

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.

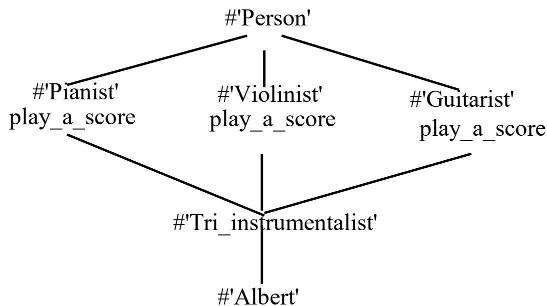

Fig. 5 Points of view of #' Albert'

For example, in fig. 5 above, #'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 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'}):

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

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

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

This prevents an arbitrary choice dictated by the chronology of class specialization 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 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')

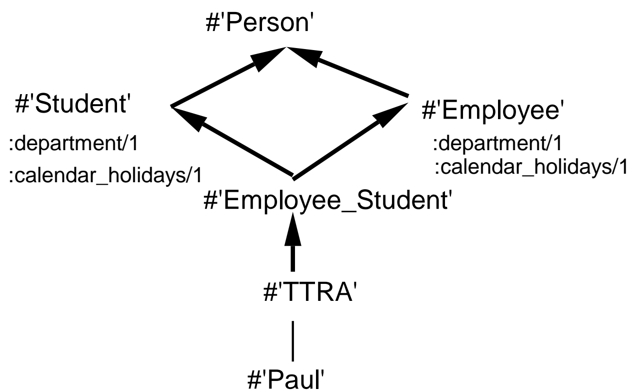

Fig. 6 Student and employee: which department/1 instance O uses at the TTRA level?

Thus, by default, OO-Prolog does not deal with method inheritance conflicts. Sending a message activates all the conflicting methods, taking advantage of the 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 (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).

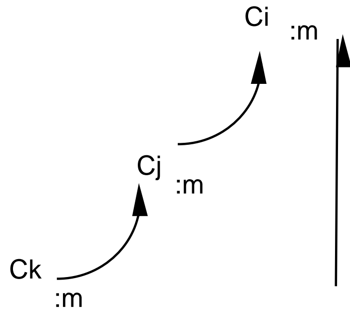

Fig. 7a Vertical backtracking

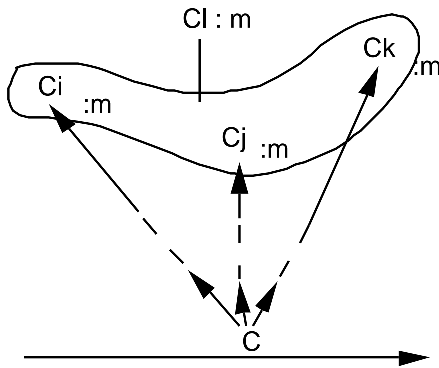

Fig. 7b Horizontal backtracking

By default, the general rule is that sending a message triggers all possible methods, taking advantage of the feedback provided by the Prolog interpreter 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. This strategy is, in our opinion, 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 class #' Student' as having a higher priority than class #' Employee'. Therefore, the object will respond to the department(D) message as a student and eventually return to its study department.

b) Linear Strategy

A 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 nature of the problems to be solved. They do not always give the same result. Therefore, the user must be given the possibility to introduce his own strategies or 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 the system will consider.

O <-- department(D).

{Val = La Haute-Seine}

A simplified version of the inversion algorithm removes 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 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.

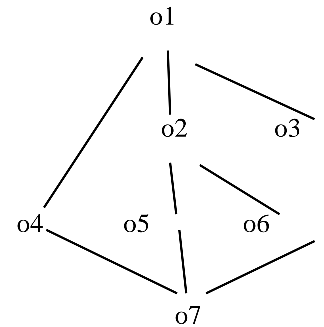

Fig. 8 Example of an inheritance graph

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.

A linear strategy is defined 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. 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 all 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 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:

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.

c) 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 allows 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 horizontal masking of the other classes. A designation may be incomplete. This is when the designated class is not defined in the property 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'.

1) Explicit Multiple Designations

In OO-Prolog, the explicit designation can be multiple, i.e. 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).

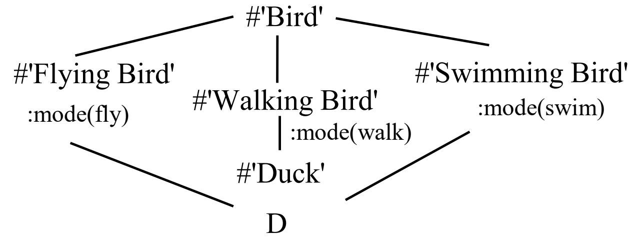

Fig. 9 Modelling the different points of view of the duck

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).

This 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.

2) 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 subclasses of the designated class.

3) 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, thus avoiding unnecessary visits to all the intermediate classes.

ACKNOWLEDGMENT

The author wishes to thank Habib Abdulrab, Jean-Pierre Pécuchet, AbdenbiDrissi-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 support for the realization of 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 (1993) 195-234.

[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, (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, St. Petersburg, Russia, 448-450 (1992).

[6] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E.Keene, GregorKiczales, 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., The object approach. concepts and tools., AFNOR, 1994.

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

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

[11] Cardelli L., A semantics of Multiple Inheritance, LNCS, Springer-Verlag, 137 (1984) 51-67.

[12] Cardelli L., Type checking Dependent Types and Subtypes, in Foundations of Logic and Functional Programming Workshop, LNCS, Springer-Verlag, 306 (1985) 44-57.

[13] Cardoso R., Mariño O., Quintero A., Corrección y completud of the multi-point classification of TROPES vista, Internal report, Computer Science department, University of the Andes, Bogotá, (1992).

[14] Capponi C., Chaillot M., Incremental construction of a type-correct class base, Actes Journée Acquisition-Validation-Apprentissage, Saint-Raphael, (1993).

[15] Chouraqui E., Dugerdil Ph., Conflict solving in a Frame-like Multiple Inheritance System, ECAI, Munich, (1988) 226-232.

[16] Clancey W.J., Heuristic Classification, Artificial Intelligence Journal, 27 (4) (1985).

[17] Cruypenninck F., Visualization interface and explanation of reasoning by classification of complex objects, Thesis of computer engineer, National Conservatory of Arts and Crafts, 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., (1988) 20-443.

[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: multiple representation and classification of objects with points of view, Doctoral thesis in Applied Sciences, Under the direction of Gérard Comyn. Supported in (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. Classificatory reasoning in a representation with multi-points of view objects. Interface homme-machine [cs.HC]. Joseph-Fourier University - Grenoble I, 1993. France. tel-00005133

[25] DUCOURNAU R., HABIB M., The Multiplicity of Inheritance in Object-Based Languages. TSI, 8(1) (1989), janvier, 41-62.

[26] DUCOURNAU R., Legacies and representations, Memory, Diploma of Habilitation to direct research, specialty: IT, Montpellier University II, (1993).

[27] DUGERDIL P., Contribution to the study of object-based knowledge representation. The OBJLOG language. Thesis from the University of 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., (1991) 245-256.

[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, (1990) 632-638.

[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) (1992) 787–803.

[32] Gallaire, H. Merging Objects and Logic Programming: Relational Semantics, Performance, and Standardization. In Proc. AAAI'86, Philadelphia, Pennsylvania, (1986) 754-758.

[33] Gandilhon T. Proposal for a minimal object extension for Prolog., Proceedings of the Logical Programming Seminar, Trégastel (mai 1987) 483-506.

[34] Gandriau, M. CIEL: classes and instances in logic. Doctoral thesis, ENSEEIHT (1988) 151.

[35] Gloess, P.Y. Contribution to the optimization of reasoning mechanism in specialized knowledge representation structures. State thesis, Univ. of TechnWorld Logie of Compiegne, January. (1990).

[36] Gloess, P.Y. M. Oros, C.M. LI, U-Log3 = DataLog + Constraints, (Prototype) Actes des JFPL95, Dijon (France), 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) (1990) 85 110.

[39] Ishikawa, Y. etTokoro, 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) (1991) 39 44.

[41] R. Jungclaus. Logic-Based Modeling of Dynamic Object Systems. Ph.D. thesis, Technical University Braunschweig, Germany, (1993).

[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) 376-385. doi: 10.1109/HICSS.1988.11828

[47] Prolog++ toolkit is an expressive and powerful object-oriented programming system that combines the best of AI and OOPs. 47. http://www.lpa.co.uk/ppp.htm

[48] Malenfant, J. ObjVProlog-V: a uniform model of metaclasses, classes and Instances suitable for logic programming, Montreal university, Dep. I.R.O., Pap. from Pech. 671 (January 1989) 58.

[49] J. Malenfant, G. Lapalme, and J. Vaucher. OBJVPROLOG: Metaclasses in logic. In S. Cook, editor, European Conference on Object-Oriented Programming (ECOOP'89), Nottingham, UK, (1989) 257-269.

[50] Malenfant, J. Design and implementation of a programming language integrating three paradigms: logic programming, object-based programming and distributed programming . PhD thesis, University of Montreal, March (1990).

[51] P. Mancarella, A. Raffaetà, et F. Turini LOO: An Object-Oriented Logic Programming Language . Proceedings of the 1995 Joint GULP-PRODE Conference on Declarative Programming (MI Sessa et M. AlpuenteFrasnedo, eds), (1995) 271-282.

[52] MARIÑO O., Classification of objects in a multi-point of view model, DEA report in computer science , INPG, Grenoble, (1989).

[53] MARIÑO O., RECHENMANN F., UVIETTA P. Multiple perspectives and classification mechanism in object-oriented representation, 9th ECAI, Stockholm (1990) 425-430.

[54] MARIÑO O., Classification of composite objects in a multi-viewpoint knowledge representation system, RFIA'91, Lyon-Villeurbanne, (1991) 233-242.

[55] MASINI G., NAPOLI A. COLNET D. LEONARD D., TOMBRE K., object languages. Inter-editions, 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, (1992) 774-779.

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

[60] B. Meyer, Reusability: The Case for object-oriented Design, IEEESoftware 4, 2 (1987) 50-64.

[61] B. Meyer. Object-Oriented Software Construction. Prentice-Hall, New York, (1988).

[62] Meyer B. Design and programming by objects, for quality software engineering, Inter Editions, 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, (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, 13-16 (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, (1999). (http://www.ci.uc.pt/logtalk/logtalk.html).

[68] A. Napoli, Object representations and reasoning by classification in artificial intelligence, Doctoral thesis in Computer Science. Defended in 1992, CRIN - Computer Science Research Center of Nancy, France.

[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), Pisa (IT) 1-9 (1992).

[70] NEBEL B., Reasoning and Revision in Hybrid Representation Systems, Lecture Notes in Artificial Intelligence, LNCS, Springer-Verlag, Berlin, 422(1990).

[71] NEWELL A., The Knowledge Level, Artificial Intelligence, 2(2) (1981) 1-20.

[72] Ngomo M., Pécuchet J-P. & Drissi-Talbi A. A declarative and non-deterministic approach to logic programming by mutable objects. Proceedings of the 4th Francophone Days of Logical Programming and Study Days Constraint programming and industrial applications, Prototype JFPLC'95, , Dijon, France (1995) 391-396.

[73] Ngomo M. , Pécuchet J-P. & Drissi-Talbi A. Managing multiple inheritance in ObjTL. RPO’95 in the Proceedings of the 15th Days Internationales IA’95,(1995) 261-270, Montpellier, France.

[74] Ngomo M., Pécuchet J-P., Drissi-Talbi A. Integration of logic programming and object-oriented programming paradigms: a declarative and non-deterministic approach. Proceedings of the 2nd Biennial Congress of the French Association for Information and Systems Sciences and Technologies, AFCET - Object Technology - 95, Toulouse, (1995) 85-94.

[75] Ngomo M. Integration of logic programming and object-based programming: study, design and implementation. Computer Science Doctoral Thesis, University of Rouen - INSA Rouen, December (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) (2018) 764-785, DOI: 10.5281/zenodo.1228893

[77] Macaire Ngomo and Habib Abdulrab, A Declarative Approach of Dynamic Logic Objects, International Research Journal of Advanced Engineering and Science (IRJAES), ISSN: 2455-9024, 3(2) (2018) 101-115.

[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, 07(05) (2018) 2036-2051.

[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, 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., Parallelization of object classification in a multi-viewpoint knowledge model, Computer science thesis, Joseph Fourier University, Grenoble, juin (1993).

[82] ROSSAZZA Jean-Paul, Use of fuzzy class hierarchies for the representation of imprecise knowledge subject to exceptions: the SORCIER system, Computer science thesis, Paul Sabatier University of 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, 97 (1988).

[85] Shapiro, E. Concurrent Prolog: A progress report. IEEE Computer 19, (1986) 44-58. (Also Chap. 5 in)

[86] E. Shapiro, (Ed.), Concurrent Prolog, 1&2(1987), MIT Press.

[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 for (sémantic) web, (2017).

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

[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, (1984) 265-270, Atlantic City, New Jersey.

[95] F. Rechenmann, A. Bensaid et D. Granier. SHIRKA: object-centered expert systems. Proceedings 4th international days on expert systems and their applications, mimeographed acts, Avignon, FR, (1984).

[96] F. Rechenmann et M.-S. Doize. SAFIR-SHIRKA: an object-centered knowledge-based system for financial analysis. Proceedings 7th international days on expert systems and their applications, Avignon, FR, (1987) 949-967.

[97] F. Rechenmann, P. Fontanille et P. Uvietta. S H I R K A: user manual. Internal report, INRIA — ARTEMIS Laboratory, Grenoble, FR, (1988).

[98] P. Barras, J. Blum, J.-C. Paumier, F. Rechenmann et P. Witomski. EVE: an object-centered knowledge-based PDE solver. Dans J. Rice, R. Vichnevetsky (éds.). Actes 2nd international conference on expert systems for numerical computing. rapport de recherché CSD-TR-963, Purdueuniversity, West-Lafayette, IN US, (1990) 1-3.