Leveraging Declarative Approaches for Efficient Dynamic Logic Object Management

Anna Kowalska[1] & Piotr Nowak[2]

[1]Warsaw University of Technology, Faculty of Electronics and Information Technology, Plac Politechniki 1, 00-661 Warsaw, Poland
[2]University of Warsaw, Faculty of Mathematics, Informatics and Mechanics, Banacha 2, 02-097 Warsaw, Poland

ABSTRACT

The marriage of logic and objects is a very wide-ranging problem, approached with various approaches, depending on the purpose. In this article, we are interested in the modelling of the state and the change of the state of an object in logic programming. After a state of the art on the subject, presenting the various aspects as well as different solutions proposed in the literature, the article then proposes a mechanism of versions of objects based on the mechanism of unification and on the use incomplete structures. Indeed, the overview of an incomplete structure can be used to allow the entry of new information by means of unification and thus to foresee the future. This mechanism makes it possible to construct the history of an object by unification and to undo it by backtracking. The changes of state are thus made and defeated, without effects of edge, in synchronization with the backtrack.

Keywords: Logic Objects, Object-Oriented Programming, Logic Programming, Prolog, OO-Prolog, Modelling, Object Condition, Object State Change, Semantics of State Change

I. INTRODUCTION

The idea of combining the aspects of object-based programming with those of logic programming dates back to the early 1980s and motivated many researchers. The goal is to take advantage of the two paradigms and reduce their respective disadvantages. Object-oriented programming has proven to be appropriate for the construction of complex software systems. On the other hand, logic programming is distinguished by its declarative charm or flavour, built-in inference and well-defined semantic capabilities. The marriage of these two paradigms can be justified in these terms and should make it possible to increase the possibilities of use, to widen the fields of application of the languages that result from it, and to lead to more efficient, more intelligent systems. These include developing complex representation and knowledge processing languages.
Logic programming provides an opportunity to formulate and solve problems declaratively. In logic programming languages, problem solving will be done by describing what needs to be done instead of describing how it should be done as long as this is the case when using procedural programming languages. The declarative way of programming offers a good method for building the software, for example for knowledge of systems, database applications, etc., because software developers must then be much less concerned with the procedural aspects of the software. their programs because they use a conventional programming language. In addition, object-oriented programming as a special programming paradigm provides benefits for software engineering. In object-oriented programming languages, the relevant world to model is considered a collection of stand-alone objects that encapsulate data and procedures. Objects are hierarchically structured and can inherit methods, namely data and procedures. This improves the reusability and maintainability of the software. Although several attempts have been made to combine both logic and object-oriented programming, the characteristics of the two paradigms have often not been met, including the declarative semantics of logic programming.

In this paper, our interest is focused on the modelling of the state and the change of the state of an object in logic programming, with emphasis on the preservation of the declarative semantics of programming in logic. This is a difficult subject in that it raises the problem of the formal semantics of updates. The article is organized as follows. In the first part we describe the different aspects of the problem and present the existing solutions. In the second part, we present a new mechanism of object versions, based on the unification mechanism and on incomplete structures. This mechanism implemented in the OO-Prolog language is then compared to other approaches. OO-Prolog is a programming language that consistently integrates programming paradigms into logic and object-based programming. It is fully developed in Prolog. In this language, an object is a named collection of Prolog predicate definitions. In this sense, an object is similar to a Prolog module. The object system is defined as an extension of the Prolog module system. In addition, an object can have attributes with values that define its history and a future that gives it a perspective of evolution in tree time. The predicate definitions belonging to an object are called methods. Thus, an object is conceptually a named collection of methods and attributes. Each object has a unique identifier. Some of the methods defined for an object should not be stored in the object explicitly, but rather are shared with other objects by the inheritance mechanism. The object system allows objects to be defined in a file, or created dynamically during program execution. In any case, during the resolution, the programs are loaded into the resolution environment. Objects defined in a file are integrated into the Prolog environment. That is, objects have a specific syntax like Prolog terms, and can be loaded into the Prolog environment. The defined objects can be either static or dynamic. In addition, the methods can be either dynamic or static. These properties are inherited by the sub-objects. Objects created during execution are dynamic. The inheritance mechanism is implemented using the import mechanism of the module system. Inheritance is a default inheritance by the overriding mechanism, which means that if a method is defined locally, and the same method is defined in a super object, then the clauses of the super method are not part of the definition of the locale, unless explicitly designating the class that defines the desired behaviour. As usual in Prolog, the methods can be undefined in a definite way, and alternative answers can be retrieved through backtracking. Using the delegation mechanism, other methods of knowledge sharing can be implemented by the user. In objects, there is a first proto-object prototype called "object", from which other objects can be constructed, directly or indirectly.

II. STATE OF THE ART

2.1 The different aspects of the problem

2.1.1 The behaviour of a logic variable

In traditional object languages (Java, C++ [Stroustrup 92], CLOS [Bobrow 88a, 88b, Steele 90], Smalltalk-80 [Goldberg 83], Eiffel [Meyer 87a, 88, 90], etc.), state of an object is represented by the values assigned to its imperative instance variables and can be modified by assigning new values to these variables. Each variable represents a memory location whose contents may change by assigning a new value. However, a logic variable represents a unique but unknown entity and not a memory location whose contents can be changed by assigning a new value. It cannot therefore substitute for a mandatory variable. Once a logic variable has been instantiated, the only way to undo its value is to go back (backtrack).

2.1.2 The intrinsic limitations of first-order logic

Another basic difficulty of this integration is that the first-order logic programming on which a large number of logic programming languages such as Prolog - the best-known and most widely used - seems to be fundamentally incompatible with the change of state. Indeed, the change of state introduces a temporal element; hence the need to look for alternative semantics. Ideally, we would like formal semantics, using, if possible, well-defined logics. Note that logic programming is not linked to a logic system like first-order logic or a language like Prolog. It groups together all the languages based on a well-defined logic system.

2.1.3 The search for a balance between theoretical and practical aspects

In object-based programming, we must propose a way to model the state of objects and introduce state changes by finding a balance between the respect of the declarative semantics and the effectiveness of the implementation mechanisms so that applications are not too penalized in terms of performance at runtime. In practice, it is always necessary to look for the best compromise between these two criteria. This goal must be achieved by providing meaningful and understandable operational semantics, based on effective inference mechanisms [Malenfant 90b] and a logic system that facilitates implementation.

2.1.4 Identification of objects

For [Bouché 94] who uses "Booch thought" [Booch 92], "an object is defined as anything that has an identity, a state and a behaviour". "The identity of an object is the property of an object that distinguishes it from all others" [Khoshafian 86]. An object behaves like a living being, whose state evolves with time, but which one can always identify, in its different forms (states). In addition to the flexibility of manipulation it offers, the identity of the objects also serves to their "modifiability". These two notions are closely related. The absence of this important property in languages like CIEL [Gandriau 88] or LOGIN [Gallaire 86] has important consequences on the semantics of state change. In particular, two equal objects (in the sense of equality of structures) will necessarily be identical since the only possible structural comparison makes them identical.

2.1.5 The influence of the order of operations on the state of objects

The behaviour of an object is influenced by its history; the order in which operations are applied to an object is full of consequences. The reason for this behaviour depends on the time and existence of a state in the object. The classic image of time, used in object systems (imperative approach of programming), is the one used in Newtonian physics. Time is a "one-dimensional linear continuum". In certain theories or modes of reasoning, we are led to use a non-linear time model, where a moment may, for example, have several futures unrelated to each other. This is true for the temporal logic that uses a tree time.

2.1.6 Semantic problems of "assert" and "retract" update predicates

Several Prolog systems offer "assert" and "retract" update predicates to dynamically modify programs and have long sought to define reasonable semantics for them [Moss 86; Lindholm 87]. These predicates have been known since their appearance as one of the gray areas of the Prolog language and their semantics are procedural. Even worse, it is not defined in a standard way. Out of twelve Prolog sites he studied in 1986, Moss distinguished nine different behaviours from the "assert" and "retract" predicates [Moss 86] and identified three major issues. First, since the "assert" and "retract" predicates result in dynamic program changes, the queries that contain them become sensitive to the order in which the goals are executed. Second, the influence of the predicates on the goals already called in the presence of the backspace; the problem of the visibility of the effects of the "assert" and "retract" predicates then arises for the goals already called. This leads to problems of consistency and program termination. Finally, the use of the "assert" and "retract" predicates raises the problem of changing the quantization of variables that occurs when dynamically adding a partially instantiated rule that can be instantiated later. Indeed, the logic variables in a query are quantized existentially while the variables in a rule are quantized universally [Warren 84; Bowen 85b]. Thus, when an "assert" adds a rule containing variables to the base, the status of these variables changes from an existential quantization in the query to universal quantization in the database. When we use the rules (facts) to represent the objects, we are then faced with this problem. Three solutions are proposed in the literature to treat this problem [Chen 88a]:

- Allow only the addition of fully instantiated facts [Warren 84]. This solution is too restrictive and therefore does not allow the modelling of situations where the programmer has only partial knowledge of the domain (the unknown being represented by free variables).
- Explicit quantification of variables [Warren 84; Machanda 88]. This solution is interesting but a bit of a constraint for the programmer who does not see his programming efforts diminish. - The management of existential variables. This solution is interesting since it allows a natural link between the variables in a query and those in the database. However, it is difficult because it poses the problem of the management of existential variables in the database.

2.1.7 Improvements in the behaviour of the "assert" and "retract" update predicates

Three major movements have shown the need to define a coherent semantics of "assert" and "retract". First, the portability requirements of large applications written in Prolog have emphasized the fact that consistency between Prolog implementations is necessary, even for predicates recognized as not having declarative logic semantics [Lindholm 87]. Then, to consistently handle updates and avoid edge effects, some Prolog systems have predicates for temporary addition of rules to the database. Finally, deductive databases, in connection with logic programming and Prolog as programming and query language, require well defined semantics of updates to improve program comprehension and reliability [Warren 84; Naish 87; Machanda 88].

2.2 Solutions for Modelling the State and State Change of an Object in Object-Based Logic Programming

Several approaches have been proposed to model the state of an object in object-based programming. In this section, we will describe the main existing proposals. The question of modelling the change of state of an object is often closely linked to the choice of an approach for the representation of its state. Thus, we will present the various modes of representation of the state of an object to discuss more precisely how is to model the change of state in each case.

2.2.1 Modelling based on imperative variables.

This approach consists of directly transplanting, in a logic programming language, regardless of the declarative semantics, imperative variables as they exist in traditional object programming. The state of an object is then represented by a set of instance variables to which values are assigned using an assignment statement. State changes are done in a destructive way, with no possibility of backtracking. This approach is essentially pragmatic and incompatible with the declarative style of logic programming. It is especially appreciated for its efficiency of calculation that by a need of proof of computation. Several languages are constructed according to this schema: ESP [Chikayama 83, 84], LOOKS [Misoguchi 84], SPOOL [Fukunaga 86], Orient84 / K [Ishikawa 86a, 86b, 87], PROBE [Gandilhon 87], Prolog Objects, etc. To clarify our point, here are two languages representative of this approach.

Objects Prolog is an extension of Prolog SICStus [SICStus Prolog, 2017]. Objects Prolog is based on the concept of prototype. In object-oriented programming, a prototype is an object that represents a typical behaviour of a certain concept. A prototype can be used as is or as a template to build other objects that share some of the characteristics of the prototype object. These objects can themselves become specialized prototypes and used to build other objects and so on. The basic mechanism for sharing is inheritance delegation. Using the delegation mechanism of an object can convey a message to another object to invoke a method defined by the recipient, but interpreted in the context of the sender. In Prolog Objects, an object is a named collection of predicate definitions. In this sense, an object is similar to a Prolog module. The object system can be seen as an extension of the SICStus Prolog module system. In addition, an object may have attributes that are editable. Predicates belonging to an object are called methods. Thus, an object is conceptually a named collection of methods and attributes. Some of the methods defined for an object should not be stored in the object explicitly, but rather are shared with other objects by the inheritance mechanism. The inheritance mechanism is implemented using the import mechanism of the module system. As usual in Prolog, the methods can be undefined in a definite way, and alternative answers can be obtained through backtracking.

Prolog++ is an APL Associates product for object-oriented programming extensions of APL Prolog [APL 2017]. Prolog++ is a complete object-oriented system integrated into a Prolog framework. Objects and instances provide a convenient way to structure related knowledge and data elements. A hierarchy of objects (or classes) makes it possible to define the information at the highest relevant level and to inherit it via the taxonomy. This distributes data and functionality along a line from general to specific. By segmenting information with this approach, complex data relationships can be efficiently managed. The ability to define object taxonomies with Prolog ++ and manipulate them with Prolog rules provides a powerful combination for serious programmers. Most Prolog ++ programs can be easily converted into Prolog Object programs.

2.2.2 Modelling based on logic terms

Another commonly used approach is to make an object a logic term also called "object-term". In logic programming, a closed functional term represents an element of the domain. It can represent structured data, analogous to a structure of a structured programming language such as C, Java, PHP, etc. The functional symbol is then interpreted as the name of the type or class. The arguments of the term represent the state of the object. In the system of Zaniolo [Zaniolo 84] and Stabler [Stabler 86], for example, the term point (Abs, Ord) represents a set of objects and has as possible instance the term point (2,3). CIEL [Gandriau 88] is another language that is based on this approach. In this language the class Point can be defined as follows:

(class **Point** {abs: Integer, ord: Integer } (methods
print(**Point** {abs: x, ord: y}) -> write(x),write(' - '),write(y);...))

An instance of this class can be defined by instantiating the arguments of this object-term: Point{abs: 2, ord: 3}.
In general, the functor of a term-object (Point, in the example above) represents the name of a class. The definition of an instance is done by instantiating the arguments of the object-term with constant values.
Based on this reasoning, several languages associate logic terms and objects and make an object a logic term, also called "object-term".

In LOGIN [Aït-Kaci, 86], LIFE [Aït-Kaci, 88, 89a, 89b, 91, 93] and U-Log [Gloess, 84, 85, 89a, 89b, 90, 91, 95] which use this mode of representation, an object is represented by a "Psi-term". For example, the Psi-term below represents the structure of the instances of the person class.

person(name => N:string, age => Age:integer, father => person(name => N))

As in the first languages, a particular instance is defined by instantiating the arguments of the Psi-term by constancy values: person(name => dupond, age => 12, father => person(name => dupond))
In this approach, state changes are made by creating a new term with new parameters. As the example below shows, CIEL uses this approach and requires the object on which a method operates to explicitly appear as an input and output argument to the method : push(stack(S),X,stack([X|S])).

In fact, CIEL is a logic programming language in which the notion of assignment does not exist.

The main consequence is that the value of a class instance can not change. To simulate the state change of an instance when applying a method, another object is created by unification. Here, the management of the consumption and the production of the terms-objects becomes the responsibility of the programmer who must take great care to pass from one method call to the other the good state of the object. This approach preserves the declarative semantics of programs and does not go beyond the framework of first-order logic. The logic object-term approach is interesting from a logic point of view insofar as it preserves the declarative semantics of logic programming. However, from the point of view of object-oriented programming, it poses several problems and is therefore often criticized:

- the difference between class and instance is not as clear as in conventional object languages.

- the identifier of an object is seen as a pointer to the structure of the object. Languages using the logic-based approach are often devoid of this important feature of objects and do not distinguish two equal but not identical objects.
- Failure to respect the principle of encapsulation.
- Syntactic verbosity: the objects being identified by their data, the user directly manipulates the whole structure of the object, with all its parameters. The number of arguments of an object-term being sometimes high; this leads to languages whose writing is characterized by syntactic verbosity.

This approach is therefore interesting, but we must look for a way to solve these different points.

2.2.3 Modelling based on atomic formulas

Conery's logic objects [Conery 87a, 87b, 88a, 88b] are another technique for using first-order logic to model objects with an editable internal state. The goal is always to introduce the advantages of object programming in logic programming so as to have a minimum impact on the existing logic programming structure (that of Prolog in particular). It is a system in which the operational semantics are defined by proof of resolution in the logic of the first order. In Conery's schema, a logic object is represented by an atomic or literal formula. In a program, the set of predicate symbols is divided into two subsets: one for object names and another for procedure names. Literals with object names as predicate symbols are called "object literals", and literals formed from procedure names are called "procedure literals". The program below contains a definition of the class "Pile" which allows illustrating these two notions. In this one, stack (ID, L) is an object literal and empty stack (ID), stack (X, ID), depilate (X, ID) and vertex (X, ID) are procedural literals.
Description of valid batteries:
A valid stack is here an empty stack or a stack whose all elements are integers.

```
stack(ID, [] ).
stack (ID, [X|L]) <-  integer(X) /\ stack (ID,L).
```

How to create a stack:
The creation of a stack consists in introducing in the resolvent (expression introduced in order to reach or complete a solution) the object literal stack (ID, []). The "new_pile" method below has a special role in the description of the "Stack" class. Its function is to introduce into the system a new object literal.
Methods of the class 'Stack':

```
emptystack(ID) /\ stack(ID, [] )<- stack (ID, [] ). % emptiness test.
headup(X,ID) /\ stack(ID,S)<- integer(X) /\ stack(ID,[X|S]).
unstack(X,ID) /\ stack (ID, [X|S])<- stack (ID,S).
top(X,ID) /\ stack (ID,[X|S ])<- stack (ID,[X|S]).
```

A query is really a pair of queries that are linked through shared variables. Part of the query concerns only literal procedures; the other consists of object literals. Complete proof requires both evidence of the existence of objects and proof that these objects satisfy a given set of constraints. Both sub-proofs are linked and executed simultaneously. Object literals are used to define objects and their states. A positive object literal (at the head of an object clause) defines the structure (name and arguments) of a class. Negative object literals (in the body of an object clause) represent class instances, where arguments are the current values of state variables. On the other hand, the procedure literals at the head of an object clause define the name and list of parameters of the methods. In the body of the clause, the procedure literals define method calls. The table below provides the procedural interpretation of a number of clauses, with various combinations of object literals and procedures.

Table 1: Procedural and declarative interpretation of the object clauses.

Clause	Declarative reading	Procedural reading
p.	p is true.	the procedure p is solvent.
s.	s is an object.	s is a valid object.
<- p.		call of the procedure p; proof of p.
<- s.		create an object with state s.
p <- q.	p is true if q is true.	to solve p, solve q.
s <- t.	s is an object if t is an object.	given an object with state s, create an object with state t; transform s into t.
p <- q / \ s.	p is true if q is true and s is an object.	to solve p, solve q and create a object with state s.
s <- q / \ t.	s is an object if q is true and t is an object.	s can be transformed into t if q is solvent.

In Table 1, p and q are literal-procedures; s and t are literal-objects. The procedural interpretation of a negative object literal is "to create an object with state s". A positive object literal is a pre-condition for the execution of goals in the body of an object clause. In other words, in the order in which the goals in the body of a clause are invoked, there must be an object that satisfies the head of the object clause. A rule with an object literal at the head and another object literal of the same name in the body (with new parameters) can then be interpreted as a state transformation rule of an object. The state changes of the objects are then made by this type of rules.

Semantically, Conery pays the high price because it allows the user not to put the object in parameter methods. He prefers to automatically process the consumption and production of object literals. This automatic processing obliges him to create a total order between the objects solved with the help of object rules, which goes against the declarative semantics which is insensitive to the order of reduction of the goals. In general, a query, consisting of a mixture of object literals and procedure literals, represents a query for proof of the existence of a set of objects and for the accuracy of a number of conditions. Since object clauses have declarative semantics, we can compare an object proof with the more conventional one.

Logic objects give assignment semantics (assignment) in terms of modified procedure proof. This allowed Conery to state that the executions of the object programs correspond to the logical consequences of the theories [Conery 88a]. Nevertheless, there is no way to give declarative semantics as good as theoretical proof. The lack of such declarative writing for objects weakens the argument that logic objects are logical. From the point of view of object-based programming, although it introduces the notion of object identifier, it suffers from a number of defects:

- The lack of structuring of programs (important aspect in programming by objects); the structure of a program is the same as that of a PROLOG program.
- As in the previous approach, the distinction between classes and instances is not clear. In addition, a class only exists by its methods. Indeed, the definition of a class is done by declaring the rules that define the behavior of its instances.
- If syntax verbosity is suppressed in the message sending protocol, it remains in the signature of a method in which the entire structure of the object must appear as a literal.
- It is difficult to define inheritance with object clauses. Indeed, a call to a literal procedure results in the call of the associated object literal (explicitly bearing the name of a class).

However, this system represents an interesting approach to representing objects with state. By modifying the proof of procedure, to allow the modelling of the assignment, we obtain a system that makes it possible to simulate the change of state rather than a purely descriptive formulation.

2.2.4 Modelling based on perpetual processes

This approach consists of modelling an object with a perpetual process defined by a recursive predicate. As in the previous approach, the predicate functor represents the name of the object. Some arguments of this predicate are intended for the representation of the state of the object. A perpetual process characterizes what intuitively causes changes over time. The change of state is then modelled by substituting for a goal-process that unifies with one of the rules of the predicate-object the goal-process network specified by the body of the rule in question. Several languages are based on this approach. These languages are often based on competing logic programming languages such as Concurrent Prolog [Chapiro 83a, 83b, 86, 87, 89], KL1 (Knowledge Language 1).

Shapiro and Takeuchi [Shapiro 83b, 87] model an object with Concurrent Prolog processes as in the example below that we have already presented and which we voluntarily resume here to illustrate this approach:

```
counter([initialize | Messages],Etat) :- counter(Messages?, 0).
counter([up | Messages], State) :-
        New_State is State + 1, counter(Messages?, New_State?).
counter ([down | Messages], State) :-
        New_State is Etat - 1, counter(Messages?, New_State?).
counter([show(State) | Messages], State) :-
        counter(Messages?, State).
counter([ ], State).          % stopping the process.
```

Here, the counter object is a goal whose behaviour is defined by a predicate and the first argument of the predicate-object is a list of messages. The rules define the behaviour of a goal-object according to the received messages. All of this happens in a competing language. One of the main problems is sharing an object between portions of the program that want to use it. The linearization of the operations is done by order of the messages rather than by that of the versions. In Shapiro and Takeuchi's approach, everything happens in a competing and deterministic logic programming language (Concurrent Prolog) that no longer assumes the important properties of logic programming as completeness. The introduction of determinism is often justified by a gain in efficiency of execution. Vulcan [Kahn 86a, 87], Mandala [Furukawa 84; Ohki 87, 88] and Polka [Davison 88, 89b, 91] are three languages in the same lineage. Since Concurrent Prolog communication management is problematic, Vulcan suggests using a pre-processor to automatically handle and polish the language syntax at the same time. Mandala is a language based on the KL1 language developed as part of the fifth generation project in Japan. Polka offers a syntax built over Parlog [Clark 86, 87], a programming language in parallel logic similar to Concurrent Prolog and which facilitates the writing of programming by objects.

LO [Andreoli 89, 90a, 90b, 90c, 91, 92] is another framework for amalgamating the paradigms of logic programming and object-based programming and which also represents an object by a predicate-process and the state of an object by the arguments of a process. Thus, as in the schema of Shapiro and Takeuchy, the dynamic behaviour of objects is then expressed, linearly, in terms of the search tree. The theoretical foundation of Linear Objects is Girard's linear logic [Girard 87, 89], a logic introduced to provide a theoretical basis for the study of competition. A major advantage of LO is to have a well defined logic as theoretical support. It thus preserves the declarative writing of logic programming. As an example, consider the class of points in the plane, with both slots x (abscissa) and y (ordinate). One possible instance of this class is (3,5). In the program below, the trans (Dx, Dy) and projx methods modify the state of a point by creating a new process with new parameters that define the new state of the object.

```
point @ [trans(Dx,Dy) | S] @ x(X) @ y(Y)
        `New_X is X + Dx, New_Y is Y + Dy
        <- point @ S @ x(New_X) @ y(New_Y).

point @ [projx | S] @ x(X) @ y(Y)
        <- point @ S @ x(X) @ y(0).
```

In the first, the state of the object goes from (X, Y) to (X + Dx, Y + Dy). In the communication stream S, the first message, that is to say the one that comes immediately after trans (Dx, Dy), will be processed in the new state (X + Dx, Y + Dy) of the object. The messages are processed linearly according to their order of appearance in the communication flow of the object. The main objections to this approach are often:

- Syntactic verbosity as in the first approach.
- The difficulty of managing the communications and in particular to share the same object between several portions of the program.
- It should also be noted that all this happens in a language that is parallel and programming in parallel logic such as that of Concurrent Prolog no longer provides important properties of logic programming as completeness[1].

2.2.5 Modeling based on logic rules

This approach consists in seeing an object as a base of rules and in representing its state by the set of rules present in this. It allows us to retain the unification of data and procedures specific to logic programming where the rules and facts use the same representation. It has the advantage of addressing the elements handled directly by logic programming, the rules, and not of interpreting them according to the concepts of object programming. The object-rule approach also results from an abstraction where the concept of the theory of logic is made to correspond to the concept of an object. This analogy leads us to consider a class as the description of a theory or meta-theory and a metaclass as a meta-meta-theory (Malenfant 90b). Several languages are based on this approach: POL [Gallaire 86], ObjVProlog [Malenfant 89a, 89b, 89c, 89d, 90a, 90b, 91, 92], Prolog ++[Moss 90, 94] [LPA 2017], etc.

In this mode of representation, an object can be seen as a theory and the change of state as the modification of this theory [Malenfant 90b]. This brings us back to the problem of the semantics of a theory whose assertions can be modified during deduction. Indeed, if an object is to be seen as a logical theory, what meaning can be given to the changes of this theory? If we admit the modification of a theory during deduction, we are confronted with the problem of the semantics of a theory whose assertions can be modified during deduction. On the other hand, the dynamic addition and removal of clauses in the database raises the problem of the consistency of updates and the change in the quantification of logical variables.

Languages such as Prolog / KR [Nakashima 84], Object-Prolog [Doma 86], Scoop [Vaucher 88], Prolog ++ [Moss 90, 94] [LPA 2017], use the Prolog assert and retract or similar predicates (eg a record example in Delphia-Prolog) whose semantics are imperative. The languages that use them suffer from the same problems of semantic order and coherence. These predicates are often preferred for their computational efficiency.

In the absence of a logical semantics, [Malenfant 90b] adopts an operational approach consisting, according to his own words, to preserve the maximum of the logic of the Horn rules and to define an operational semantics of the changes of state of the objects which limit the effects on the semantics declarative. In the "object version mechanism" it proposes for the implementation of the ObjVProlog-V (ObjVProlog with Versions) language [Malenfant 90b], the object versions subdivide an object into a sequence of rule bases. A resolution context is then a triplet (<object>, <version>, <class>), where <version> and <class> respectively indicate the rule base in the sequence that forms the object and the level in this base rules. Four rules then make it possible to determine in which version a goal must be solved [Malenfant 90b]. According to this approach, an object is built of a sequence of versions that represent the history of state changes for that object since its creation. Thus, when a change is executed, conceptually, a new copy of its rule base is made. A message to an object is normally fully resolved in the context of the latest version of the object in this rule base when the object begins to resolve it. Contrary to the approach we advocate, ObjVProlog-V's object versioning mechanism is a mechanism that seeks to separate as much as possible the backtracking, to find solutions to a message, the classic behavior associated with the change of state of the objects. As a result, the state change is seen as a behavior that is not related to backtracking. [Malenfant 90b] justifies this choice by the fact that the change of state for the objects usually implies a progression in time which is badly related to the backtracking.

2.2.6 Modeling based on intentional variables

Chen and Warren [Chen 88a] have addressed the problem of logical programming assignment by proposing to use Montague's intentional logic as a semantic basis for changing values of variables. Intentional variables are modelled as a sequence of values in each state, and during deduction, goals are solved in a given state as long as there is no change of state. The deduction procedure with intentional variables makes and breaks the state changes in synchronization with the backtracking. This approach has a clear semantics in intentional logic. It should serve as a well-defined semantic alternative to imperative variables.

```
        vide(IP) :- IP :: [].
        top(IP, X) :- IP :: [X | _].
        stacking(IP, X) ::= IP :: Stacke, IP := [X | Stacke].
        unstacking(IP) ::= IP :: [_ | Stacke], IP := Stacke.
```

As the example above shows, there are two types of predicates:
- static predicates, defined by static rules introduced by the ": -" operator;
- and dynamic predicates, defined by dynamic rules introduced by the ":: =" operator.

The interpretation is as follows. A static rule is identical to a Horn rule except that it may contain access to the value of an intentional variable, represented here by the operator "::"/2. A dynamic rule allows you to modify an intentional variable using the operator ":"/2.

2.2.7 SWI-Prolog approach for web semantic

The Web (semantics) is one of the most promising areas of application for SWI-Prolog. Prolog manages the natural RDF semantic web model, where RDF provides a stable model for representing knowledge with shared semantics. It turns out that Prolog is also quite capable of providing web services (HTTP), especially where it comes to dynamic generation of HTML pages and providing data for JavaScript in web applications by using serialization JSON. This is an imperative approach that does not respect the declarative semantics of logic programming.

2.2.8 Other approaches

LOO [Mancarella 195] is an object-oriented language in logic programming. The Loo language combines object-oriented programming with logic programming. Authors define model classes as sets of clauses that represent their methods. An object is an instance of a class and is identified by a unique name. They use a set of operators on theories of manipulation of state changes and for the inheritance of modelling. The authors remain very vague and give no details on the modelling and implementation of these mechanisms. A message sent to an object results in an objective that is resolved relative to a dynamic composition of clauses representing its class and its current state. The challenge is to avoid superimposing a complex syntactic and semantic structure over the simple structure of logical programming. The authors say they have tried to extend logical programming in a conservative way, as much as possible, in order to maintain simple and clear semantics.

2.2.9 Comparison of approaches

This multitude of approaches shows the wealth of logic programming that offers several formalisms of representation. With the exception of models based on imperative variables, all the others manipulate elements of logic programming (logical term, predicate, logical rules, etc.). However, their level of granulity differs.
In the logic-based approach, one essentially seeks to interpret the elements of logical programming in terms of object-oriented programming. By confining itself to interpreting terms as objects, some advantages of logical programming can be made to object-based programming, but relatively little is made of logical programming [Malenfant 90b].

Clauses and predicates completely change perspective on terms. In fact, it changes the way data structures and terms are handled, much like a typing system does. The clauses are at a level of granularity where one is not interested directly in the terms that are manipulated by rules, but in the sets of clauses seen as largely autonomous bases of knowledge, behaving like logical programs.

However, if the programming with the clauses of Horn has a clear semantics, this representation mode poses the problem of updating the base of clauses during the resolution (change of quantification of the variables, coherence of the updates, etc.).

Approaches based on logical terms, literals and perpetual processes do not experience the same semantic problem as rule-based approaches. The defects noted come rather from the non respect of certain characteristics of the programming by objects. Approaches based on logical terms, literals and processes generally suffer from syntactic verbosity. In these approaches, the objects are devoid of identifier and are identified by their structure.
We also note that in these approaches, the distinction between classes and instances is not as clear as in conventional object languages. In some of these approaches (Logical objects of Conery, objects in Concurrent Prolog, LO, etc.), a class exists only by its methods.

These approaches, however, offer advantages in terms of unification and have clear logical semantics. In particular, they provide a solution to the problem of changing the quantization of logical variables since, during the resolution, all the actions on the objects are performed in an existential environment.
In Vulcan, SCOOP, 'Objects as Intensions', ObjVProlog, the atom that represents the object identifier is generated by the system to ensure its uniqueness.

After having reviewed the main existing approaches in logic programming to model the state and the change of state, we note, despite the multitude of proposed solutions, the difficulty of establishing state changes of objects on a semantic logic and effective implementation mechanisms. The search for a logical framework for the semantics of state changes of logical objects and that of implementation mechanisms remains, from this point of view, a very open subject in that, the answer to all the considerations , theoretical and pragmatic, which constrain the definition of a programming language in logic and object-oriented is not easy. In the next section, we describe our approach to modelling state and state change in logic programming. Our approach takes into account both declarative semantics and the effectiveness of implementation mechanisms.

III. THE LOGIC OBJECT VERSION MECHANISM OF THE OO-PROLOG LANGUAGE

Classically, the change of state of an object implies a progression in time (linear time) which is badly related to the backtracking. Consequently, the change of state is seen as a behaviour that is not linked to the backtracking. As we said before, the image of time is here that used in Newtonian physics. Time is a one-dimensional linear continuum. The mechanism we propose is based on the unification mechanism, as a matching tool, and on the backtrack. Our goal is to have dynamic objects that can be built by unification and undone by backtracking. In order to avoid edge effects, we propose to manage objects in a temporary existential environment. This facilitates the links between variables. An immediate consequence is that during the deduction, the quantification of the variables involved does not change. An object can be partially instantiated. In other words, its state can contain variables that can be instantiated later. In an edge effects programming style, the focus is on a global environment. In the OO-Prolog [Ngomo 96] language, this global environment is erased by considering it as an additional parameter of each method which calculates, in addition to the normally expected result, a new environment. Objects are handled through this environment. A state of this environment represents an aspect of the universe at a given point in the time of deduction. In the OO-Prolog language, an object is characterized by its history and behaviour [Lieberman 86]: the future is represented by the set of free variables (anything can happen), the past is instantiated (it's too late) . During the deduction the objects are built by unification and defeated by backtracking. As the resolution time goes back when looking for new branches leading to new solutions, this is translated operationally by the restoration of the previous states when there is backtracking.

3.1 Internal representation of an object

In the OO-Prolog language, objects can be statically declared in a program, but dynamically manipulated via an existential, temporary and scalable environment. An object environment is represented by an incomplete structure[2] shared by all objects. This environment is a common knowledge base for objects. It has the following form: ENV = [next(PtrObject),..., clock([0|NextDate]),date(0),Id_1:E_1,Id_2:E_2,...,Id_n:E_n].

We are talking here about a dynamic environment open to changes of state. Otherwise, the environment may be static or closed, with no possibility of state changes. This is then expressed as follows:

ENV = [next(closed),..., clock([0]),date(0),Id_1:E_1,Id_2:E_2,...,Id_n:E_n].

This type of environment is used in particular for representing static knowledge (static programs) and solving problems by simply querying the knowledge base.

In this representation, each state of the environment has a date corresponding to its date of creation, date (Date) of state changes. This is then expressed as follows: Id_1:E_1, Id_2:E_2, ..., Id_n:E_n are objects present in the environment. The PtrObject parameter is a pointer to the object that will be created later. Each environment of objects is provided with a clock that contains the different moments of the evolution of the environment. The instantiated part of this environment represents the past state of the base while its uninstantiated part represents its future state, which may contain future modifications. Each E_i is also represented by an incomplete structure of the form: **E_i = [next(NextState), status(_), date(0), att_1 := val_1,..., att_i := val_i,..., att_n := val_n]**

where NextState is a pointer to the future state of the object. The "status" attribute is used to define the status of the object. When associated with an uninstantiated variable, "status (_)", the object is active. To give an object an inactive status it is enough to instantiate this variable in the following way "status (off)".

Status changes are made and defeated in sync with Prolog's backtrack. It is therefore possible to return, by backtracking, the previous states of an object or the environment of objects.

The universe of objects is formed by a series of layers ordered in time that each reflect an aspect of the universe at a given moment. Each layer only stores the information that differentiates it from the previous layer. Each object retains its history by memorizing the changes made from its creation to the present moment. By default, as in the approach of "intentional objects" [Chen 88a], the most recent version hides the old ones ("non-monotonicity").

3.2 Update Operations

The universe of objects is formed of a series of layers that each reflect an aspect of it at a given moment. The layers are ordered in time (resolution time), each memorizing only the information that differentiates it from the previous layer. There is no duplication of data. A user can operate on the most recent layer (including the previous ones), or on an earlier layer, by explicit designation. The universe of objects is represented by an incomplete structure that contains its different layers. Each object retains its history by memorizing the changes it has undergone since its creation until now. In imperative object languages such as C++, Java, etc., only the last state is usually retained, and the computer variables associated with the attributes of the instance are assigned during the lifetime of the object, without any possibility. back on the previous states of this object. In the OO-Prolog language, an object is characterized by its history and behaviour. Although the entire history of an object is available, you can access by default only the last state, that is to say the most recent, as in the example below.

```
    ?-...., P <- (setval(x(_),5), setval(x(_),10), getval(x(_),X)).
    {...,X = 10}
```

Each change made during the deduction is automatically defeated by backtracking. As the resolution time goes back when looking for new branches leading to new solutions, this is translated operationally into OO-Prolog by the restoration of the previous states when there is backtracking. Time then has a tree structure.

```
?-....,P <- ( setval(x(_),5),(setval(x(_),10);setval(x(_),20)),getval(x(_),X).
    {..., X = 10}
    {..., X = 20}
```

In order to allow access to any version of an object, each version is completely characterized by its creation time. Implantation is done using a temporal mechanism. A global time clock for creating versions is initialized to zero at the beginning of the deduction. It is incremented by one unit at each change and decremented during a "backtracking".

```
?-....,P <- ( setval(x(_),5,T1),
  (setval(x(_),10,T2);setval(x(_),20,T2)),getval(x(_),X,T1),  getval(x(_),Y,T2) ).
{..., T1 = ..., T2 = ..., X = 5, Y = 10}
{..., T1 = ..., T2 = ..., X = 5, Y = 20}
```

We see in this example that the value of the abscissa of P is 5 at time T1 and 10 or 20 at time T2. Time is manipulated here explicitly.

Dynamic creation of an Id_{n+1} object consists in adding the Idn + 1 object in the uninstantiated part of the object environment. Suppose that ENV = [next(**F), clock([0|_])**,...,date(0), Id_1:E_1, Id_2:E_2,...,Id_n:E_n] is the state of the environment before creating the Id_{n+1} object. So after creating this object, ENV becomes:

ENV = [next(**[next(F'),date(1),Id_{n+1}:E_{n+1}]**), **clock([0,1**|_]), date(0),Id1:E1,...,Id_n:E_n].
with E_{n+1} = [next(_),date(1),...]. It's as if all other objects have been duplicated. However, we can see that there is no redundancy. This creation is carried out by the methods newObject (O, E) (formerly denoted new) and newCObject (O, E) (formerly denoted create) whose effect is to create the object O with the state E. The newCObject method (O, E) has the effect of automatically creating and classifying the newly created object.

Example:

```
?- #'Point' <- newObject(P,[]), P <- display.
TERMINAL :: < #[#'Point', 5] >
                class(#'Object') <- #Point
                x(#'Point') <- 0
                y(#'Point') <- 0
  {P = #[#'Point', 5]}
```

3.2.2 Assigning a value to an attribute

The assignment operation is to give a value to an attribute. In the OO-Prolog language, this operation is reversible because it is possible to return, by backspace, on the previous states of an object. When an attribute Att, having the value Val, receives a new value NVal, instead of overwriting the old value, as in the imperative approach of the programming, one saves the new value in the uninstantiated part of the structure representing the state of the object. Let's illustrate this procedure with a simple example. Consider the state of a point on the plane P at time 0.

E = [next(**X**), statut(_), date(0), class(#'Object') := #'Point', x(#'Point') := 1, y(#'Point') := 2]
After assigning the value 3 to the attribute x (# 'Point') of P, E is modified as follows:
E = [next(**[next(X'), date(1),x(#'Point') := 3])**, statut(_), date(0), class(#'Object') := #'Point', x(#'Point') := 1, y(#'Point') := 2].

After performing this operation, we obtain another state of the object P corresponding to the date date (1). The assignment is performed by the setval (Att, Val), setval (Att, Val, Date), setvalc (Att, Val), setvalc (Att, Val, Date) methods that take an attribute and a value as input. possibly returns the date corresponding to the creation of a new state of the modified object. The only difference between setval and setvalc (formerly setv) is that the application of setvalc to an object is followed by an automatic classification of that object. In both cases, there is control of the type of the value Val passed as argument of the method. With the initial state of our environment above, we have:

```
?- P <- setval(x(I),3,Date).
{P= #[#'Point',1], I = #'Point', Date = 1}
```

and of course we can also have, as in Prolog

```
?- P <- setval(x(_),3,1).
{}
```

which leads to a success.

There are four methods to access the value of an attribute:

```
getval(Att,Val), getval(Att,Val,Date), getv(Att,Val), getv(Att,Val,Date)
```

Example:

```
?- P <- (setval(x(_),3,Date), getval(x(_),X,0), getval(y(_),Y)).
{P= #[#'Point',1], X = 1, Y = 2}
?- P <- (setval(x(_),3,Date), getv(x(_),X,Date)).
{P= #[#'Point',1], Date = 1, X = 1}
{P= #[#'Point',1], Date = 1, X = 3}
```

As in Prolog, access operations to the value of an attribute can be used to assign, unification, a value to an attribute, if the initial value of the attribute at the given time is a free variable.

```
?- ..., P  <- ( setval(x(_),Val), getval(x(_),3) ).
{..., P= #[#'Point',1], Val = 3}
```

3.2.2 Deleting a value from an attribute

The operation of deleting a value to an attribute is defined as the assignment of a variable not instantiated to this attribute. This makes it possible to cancel the previous value on the same date and thus define a future for this variable. The attribute can thus be considered as having no value yet.

3.2.3 Rules for optimizing the global clock management process

In order to optimize the management of global clock changes and dates, we have introduced the following rules:

- The global clock can be incremented only when a change of state affects that concerns a variable already instantiated;
- The incrementation of the dates can take place only during a change of value of an already affected variable.
- In other cases, the global clock remains stable and undergoes no change.

Let's illustrate these rules with a simple example. Consider once more the state of a point on the plane P at time t = 0.
E = [next(**X**), statut(_), date(0), class(#’Object’) := #’Point’, x(#’Point’) := 1]
After assigning the value 3 to the attribute x (# 'Point') of P, E is modified as follows:
E = [next(**[next(X’), date(1),x(#’Point’) := 3]**), statut(_), date(0), class(#’Object’) := #’Point’, x(#’Point’) := 1].

Consider now the assignment of the value 2 to the attribute y (# 'Point') of P. E is then modified as follows:
E = [next(**[next(**next(**[next(X’’), date(1),y(#’Point’) := 2]**)), **date(1),x(#’Point’) := 3]**), statut(_), date(0), class(#’Object’) := #’Point’, x(#’Point’) := 1].

This time, the clock does not change, so the change only affects an attribute that was not instantiated on the current date.

This same behavior is preserved when the change is on an earlier date. Thus, assigning the value 5 to the x (# 'Point') attribute of P on date 0 will not change the global clock as shown in the code below. The environment E is then modified as follows:
E = [next(**[next(**next(**[next(**next(**[next(X’’’), date(1),x(#’Point’) := 2]**)), **date(1),y(#’Point’) := 2]**)), **date(1),x(#’Point’) := 3]**), status(_), date(0), class(#’Object’) := #’Point’, x(#’Point’) := 1].

Since the x attribute was not yet assigned to date 1, the state change made does not change the global clock.

3.2.4 Implantation

The version mechanism described above is also the one used in the ObjTL language (a prototype whose various extensions led to the realization of the OO-Prolog language) [Ngomo 95a, 95b, 95c]. However, in ObjTL the object environment appears explicitly both in the signature of a method and in the protocol of a message sending:

- Definition of a method:
  <class> << **Env** >> <selector>(<arg$_1$>,...,<arg$_n$>) :- <body>.
- Sending message:

- ***<object> << Env <- <message>*** sends goal B to object <object>, the search for the method begins at its instantiation class.
- ***<object> as <class> << Env <- <message>*** sends the goal <message> to the object <object>, the search for the method begins at the class <class>.
- ***<object> << Env <- <message>*** sends goal B to object <object>, the search for the method starts at its instantiation class and the search strategy is a linearization.
- ***<object> as <class> << Env <- <message>*** sends the goal <message> to the object <object>, the search for the method starts at the level of the class <class> and the search strategy is a linearization.

The explicit use of the object environment by the user can be a source of problems:

- a rather heavy syntax compared to the conventional syntax;
- there is probably a risk of the user manipulating the object environment directly without going through the appropriate methods.

It therefore seemed useful to polish this syntax by relieving the user of the management of this environment. This allows for a simpler syntax that is closer to conventional syntax.

- Definition of a method: The form of the object clauses becomes:

    <class> :: <selector>arg$_1$>,...,<arg$_n$>) :- <body>.

- Sending message: a message sending in one of the following forms:

- <object> <- <message> sends goal B to object <object>, the search for the method starts at its instantiation class.
- <object> <- (<class>: <message>) sends the goal <message> to the object <object>, the search for the method starts at the level of the class <class>.
- <object> <- <message> sends goal B to object <object>, the search for the method starts at its instantiation class and the search strategy is a linearization.
- <object> <- (<class>: <message>) sends the goal <message> to the object <object>, the search for the method starts at the level of the class <class> and the search strategy is a linearization.

Example: For the class of the points of the plane we will be able to define the method of access to the value of the attribute x (#'Point') as follows:

```
#’Point’ :: getx(X) :- self <- getval(x(#’Point’),X).
```

A query can then be:

```
?- #’Point’ <- newObject(P,[x(_):=2,y(_):=3]),P <- getx(X).
```

This result is obtained by meta-interpretation. Let A be the query to be reduced. Query A can contain standard Prolog literals or object-literals (sending messages). Among classical literals, we will distinguish those associated with system predicates and others. They will be solved by the interpreter Prolog. Similarly, object literals associated with the basic methods will be treated differently. They will be solved by a low level interpreter.

3.2.4 The formal unification of Prolog terms

The heart of the computational model of logic programs is the unification algorithm. Unification makes it possible to determine, if it exists, the common instance of two terms. Unification is at the root of most automatic deduction work and the use of logical inference in artificial intelligence.

A term t is a common instance of two terms t1 and t2 if there are substitutions □1 and □2 such that t equals □1t1 and equal to □2t2. A term s is more general than a term t if t is an instance of s, but s is not an instance of t. A term s is an alphabetical variant of a term t if both s is an instance of t and t is an instance of s. A two-term unifier is a substitution that makes the terms identical. If two terms have a unifying unit, we will say that they unite. There is a close relationship between unifiers and common instances. Any unifier determines a common instance, and conversely any common instance determines a unifier.

A more general unifier or "upg" of two terms is a unifier such that the associated common instance is the most general one. If two terms unite then there is a single more general unifier. This uniqueness is to "rename" variables closely. Equivalently, two univariate terms have a single most general common instance, an alphabetic variant.

A unification algorithm calculates the most general unifier of two terms, if any, and displays "failure" otherwise. The algorithm for unification presented below is based on the solution of equations. The input for the algorithm consists of two terms, T1 and T2. The output of the algorithm is the "upg" of the two terms if they unify or "fail" if they do not unite. The algorithm uses a stack to store the equations to solve and a location to group the substitution of the output.

The vast majority of Prolog systems do not use the classic unification algorithm, deliberately choosing not to perform the occurrence test (a variable can be unified to a term containing it). This choice is not without problems at the theoretical level, since it defies the model of the universe of Herbrand limited finite terms [Herbrand 67]. At a more operational level, the implementation without special precautions of such an algorithm, ignoring the test of occurrence, makes it subject to loops. Thus, unlike the original unification algorithm [Robinson 65], a variable can be linked to a term containing it. The main reason for this omission is a significant gain in execution time. Indeed, performing the test of occurrence is an expensive operation since for each substitution creation {(x, t)}, it makes it necessary to go completely through the term t in order to determine whether the variable x is or is not present in t.

In OO-Prolog, an object has a unique identifier that distinguishes objects. Two different objects can not have the same identifier. In this case, the application of the Prolog unification procedure to two OO-Prolog objects will always result in a failure since two distinct identifiers can never be unified. So we have to modify the classic procedure of Prolog unification so that it takes into account the objects and the inheritance relation.

In order for the object layer to react homogeneously with the rest of the Prolog language, it must have mechanisms identical to those of all the types of data present in Prolog. We propose to define a specific mechanism to take into account the objects.
In the case of objects, the use of this primitive poses problems of names referencing the objects. Two structurally unified objects can have different identifiers. There is no valid justification for accepting a success for the unification of different object names while the unification of distinct functor terms fails even if all the other elements composing the terms are identical [Cervoni 94]. As a result, we are obliged to have specific operators for object names. In OO-Prolog the name of an object is preceded by the operator #: #<name>. For example "#'Point'" instead of 'Point'. This notation distinguishes object names from other Prolog terms.

3.2.5 Abstract Interpreter for OO-Prolog Programs
The abstract interpreter for OO-Prolog programs is an extension of the Prolog interpreter to logical objects. This interpreter is a modification of the abstract interpreter for Prolog [Sterling 90] programs. It gives the solution of a question G relating to a program P. The output of the interpreter is an instance of G, if a demonstration of such an instance is found, or "failure" if there was failure during of the calculation. If non-object literals are reduced in the traditional way, reducing object literals requires additional processing to accommodate inheritance. Thus, if the current goal is, for example, of the form O <- M, then this literal can not be unified with any clause in P. The processing consists of finding the class of the object receiving the message, C, and browse the subgraph of C to search for the definition class or classes of method M. For each class found, there is a (renamed) clause C :: M '<- B1, ..., Bn, n ≥ 0. Once such a clause is chosen, the processing continues as in the classical case by replacing in the resolvent the current goal by the body of the clause B1, ..., Bn. Then, we apply not only to the resolvent and to G, but also to E, an incomplete dynamic structure that undergoes unifications during processing.

3.2.6 Unification Algorithm Extensions
The unification of two terms of the same class mainly consists of recursively unifying the fields of the structures of the instances. In the case of objects, two instances of the same class are semantically univariable if and only if the values of their respective attributes are uniformable in the sense of Prolog or semantically uniformable. This definition does not allow for example to unify a rectangle of length 4 and width 4 to a square of side 4, unless we use the classification mechanism. It is still necessary that the user explicitly express the classification constraints. If the instances are not of the same class, it is necessary to search between them for a possible inheritance relation which would allow to unify them, by specialization or by generalization. Extensions to the algorithm presented previously are described in [Ngomo 96]. We do not describe them in this article that focuses on the dynamic nature of objects.

3.2.6 Unification Algorithm Extensions
The unification of two terms of the same class mainly consists of recursively unifying the fields of the structures of the instances. In the case of objects, two instances of the same class are semantically univariable if and only if

the values of their respective attributes are uniformable in the sense of Prolog or semantically uniformable. This definition does not allow for example to unify a rectangle R = (length: 4, width: 4) to a square C of side equal to 4, unless we use the classification mechanism. It is still necessary that the user explicitly express the classification constraints. If the instances are not of the same class, it is necessary to search between them for a possible inheritance relation which would allow to unify them, by specialization or by generalization. Extensions to the algorithm presented previously are described in [Ngomo 96]. We do not describe them in this paper that focuses on the dynamic nature of objects.

3.2.7 Some properties of the model

3.2.7.1 A simple syntax
The OO-Prolog language has a simple syntax similar to that of conventional object languages.

3.2.7.2 Changing the quantification of variables
In OO-Prolog, the problem of changing the quantization of logical variables is solved by using existential environments, in which all variables are quantized existentially. Thus, the following queries

```
?- #'Point' <- new(P,[]), P <- setval(x(_),X,D), X = 5, P <- getval(x(_),Y).
?- #'Point' <- new(P,[]), X = 5, P <- (setval(x(_),X,D), getval(x(_),Y)).
```

both lead to the same result: {..., X = 5, Y = 5}.

3.2.7.3 Consistency of updates
The approach presented here allows you to manage updates consistently. In contrast to imperative languages that use Prolog's "assert" and "retract" predicates (such as Prolog++ [Moss 86, 90, 94] [LPA 2017]) or that implement imperative variables (such as ESP [Chikayama 83, 84]), changes in OO-Prolog are made and undone in synchronization with the backtracking. This concerns all creation, modification and deletion operations. We can then have:

```
? #'Point' <- new(P,[]), P <- ( (setval(x(_),2,D) ; setval(x(_),5,D)), getval(x(_),Val),delete(x(_) ).
{..., D = 1, Val =2}
{..., D = 1, Val =5}
```

3.2.7.4 Formal significance of updates
During the evolution of the universe of objects, each state or layer corresponds to a given moment. Let E be the set of these states and R_p be the temporal precedence relation between two states of the universe E: << for all e_t and $e_{t'}$ comparable elements of E (t and t' being two points of the resolution time, e_t and $e_{t'}$ are respectively the state of the universe at time t and at time t'), then: (e_t R_p $e_{t'}$) or ($e_t = e_{t'}$) or (e_t R_p $e_{t'}$) >>. In this case, if e_t R_p $e_{t'}$, then $e_{t'}$ inherits somehow from e_t. Since the resolution time is arborescent, the relation R_p is a partial order relation since we can not necessarily compare two elements of E. Our temporal model M is then composed of the set E, the binary relation R_p on E and a function I: E x {Formulas of language} {1 , 0} which associates to each formula of language its values of truth to the different possible states of the universe.

The interpretation of a formula is then done relative to a given state of the universe of objects, considering an interpretation as a couple (M, e). If e_i is a version before e_j then e_j somehow inherit e_i. Each copy generated contains locally only the information that differentiates it from its generator. The rest is somehow inherited. The information is stored in this structure without redundancy. The model of time is here an "finitary infinite" tree, that is to say a tree whose each node admits a non-zero finite number of successors. The sequence corresponds to the particular case where this number is worth the unit. In a strictly temporal interpretation, the sequence of situations represents the evolution of the state of the world over time.

IV. CONCLUSION AND PERSPECTIVES

State and state change modeling of an object is a central problem in object-based logic programming. This article presents an in-depth discussion of existing approaches. He then proposes a new mechanism for object versions. This mechanism is based on the unification and use of incomplete structures that are inherently dynamic and thus represent the dynamic aspects of logical objects. He uses unification as a matching tool. As a result, the state changes are made and undone in synchronization with the backtracking. An object is then characterized by its behavior and its history. The set of states is the set of versions of an object. These versions are ordered according to a partial order which expresses the successive derivations of a version, and one speaks about tree of versions. One of the problems often encountered in versioning models is the consistency of the versions between them and its maintenance through consistent configurations. The versioning mechanism of the OO-Prolog language offers several advantages, both theoretically and practically:

- compared to imperative approaches that introduce edge effects programming, it has the advantage of having a declarative, clear and coherent semantics;
- the changes are expressed in terms of a search tree, that is to say a dynamic structure of logic programming;
- the deletion of a value at an attribute on a given date from the global clock corresponding to the assignment of a variable to this attribute, which opens an evolution perspective to this attribute (a future);
- thanks to the use of an existential environment with an always available future, it facilitates the links between variables; which brings a solution to the problem of changing the quantification of variables;
- in relation to the interaction between version management, the identity of an object and the type of an object, OO-Prolog adopts a dynamic solution. An object O can reference an object O '. If the O object has multiple versions, the reference is dynamic and interpreted when the program runs. A dynamic reference can be considered as a query on all versions.

Our work continues in optimizing the implementation techniques of the proposed mechanism. The objects being manipulated in the dynamic space, this can quickly lead to a saturation of the batteries of this space. It is then necessary to limit as much as possible the write accesses in this environment which, with respect to the code zone, is much more limited. This is not a pressing need, given the current power of computers and their storage capacity. We simply want to increase the performance of our language. Our work also focuses on the design of a multi-tier architecture and service-oriented database query interface in OO-Prolog, with applications in several domains. Another avenue explored is the design of a service platform around the OO-Prolog language: interrogation service, resolution service, exchange service with other languages, etc.

V. REFERENCES

[1] **[Aït-Kaci 86]** Aït-Kaci, H. & Nasr, R. "LOGIN: A Logic Programming Langage with Built-in Inheritance". J. of Logic Programming 3, 3 (Oct. 1986), pp. 185-215.

[2] **[Aït-Kaci 88]** Ait-Kaci, H. & Lincoln, P. "LIFE, A Natural Language for Natural Language". MCC Technical Report Number ACA-ST-O74-88, Austin, Feb. 1988.

[3] **[Aït-Kaci 89a]** H. Aït-Kaci, "LIFE an overview", Presentation au Groupe AFCET-Prolog, 1989.

[4] **[Aït-Kaci 89b]** Aït-Kaci, H. and Nasr, R., P. Lincon and D. Plum, "LIFE an overview", DEC Paris Research Laboratory, 1989.

[5] **[Aït-Kaci 91]** Ait-Kaci, H. & Podelski, A. "Towards a Meaning of LIFE". Proc. of the Thirsd Int'l Conf. on Programming Language Implementation and Logic Programming, Lectures Notes in Comp. Sciences, Passaü, Aug. 1991.

[6] **[Aït-Kaci 93]** Ait-Kaci, H. & Podelski, A. "Towards a Meaning of LIFE". Journal of Logic Programming, 16:195-234, 1993.

[7] **[Alexiev 93]** 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.

[8] **[Andréoli 89]** Andréoli, J-M. & Pareschi, R. "Logic programming with sequent systems : A linear logic approach". In P. Schroeder-Heister, editor, Intl. Workshop on Extensions of Logic Programming, number 475 in LNAI, pages 1-30, Tübingen, Germany, 1989.

[9] **[Andréoli 90a]** Andréoli, J-M. & Pareschi, R. "Linear Objects: Logical Processes with Built-in Inheritance". In 9th Conf. on Logic Programming, Jérusalem, Israel, 1990.

[10] **[Andréoli 90b]** Andréoli, J-M. & Pareschi, R. "Linear objects: Logical processes with built-in inheritance". In D. Warren and P. Szeredi, editors, Intl. Conf. on Logic Programming (ICLP'90), pages 495-510, Jerusalem, Israel, June 1990. MIT Press.

[11] **[Andréoli 90c]** Andréoli, J-M. & Pareschi, R. "LO and behold! Concurrent Structured Processes". In ECOOPOOPSLA '90, Ottawa, Ontario, 1990. (SIGPLAN Notices, 25(10):44-56, Oct . 1990).

[12] **[Andréoli 91]** Andréoli, A. & Pareschi, R. "Linear Objects: Logical processes built-in inheritance". New Generation Computing, 9(4):445-473, 1991.

[13] **[Andréoli 92]** 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.

[14] **[Bobrow 88a]** D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, & D.A. Moon. "Special Issue, Common Lisp Object System Specification, X3J13 Document 88-002R". ACM SIGPLAN Notices, 23, Sep. 1988.

[15] **[Bobrow 88b]** D.G. Bobrow, K. Kahn, G. Kiezales. "The Common Lisp System Metaobject Kernel", A Status Report. In Proc. of the 1st IWOLES, pages 27-32, Paris, 1988.

[16] **[Bonner 93]** A. Bonner and M. Kifer, "Transaction logic programming (or, a logic of procedural and declarative knowledge). In Intl. conf. on Logic Programming (ICLP'93), pp 257-279, Budapest, Hungary, 1993.

[17] **[Bonner 94a]** A.J. Bonner and M. Kifer "A General Logic of Stage Change". Technical Report, Computer Systems Research Institute, University of Toronto, 1994.

[18] **[Bonner 95]** A.J. Bonner and M. Kifer "Transaction Logic Programming (or a logic of declarative and procedural knowledge)". Technical Report CSRI-323, University of Toronto, April 1995. ftp://csri.toronto.edu/csri-technical-reports/323/report.ps.Z.

[19] **[Booch 92]** G. Booch “Object Oriented Design with applications” The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1992.

[20] **[Bouché 94]** Bouché M., "La démarche objet. Concepts et outils.", AFNOR, 1994.

[21] **[Bowen 85]** Bowen, K.A. et Weinberg, T. A Meta-Level Extension of Prolog, IEEE Inttl Symp. on Logic Prog. 'B5 (1985), pp.48-53.

[22] **[Brachman 85]** Brachman R. J. and Schmolze J. G. "An overview of the KL-ONE Knowledge representation system",. Cognitive Science, 9(2):171-216, 1985.

[23] **[Cervoni 94]** L. Cervoni “Méthodologies et Techniques de résolution de Problèmes avec Contraintes. Application en Programmation Logique avec Objets : CooXi.” Thèse de Doctorat Nouveau Régime, Université de Rouen, juillet 1994.

[24] **[Chen 88]** 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.

[25] **[Chen 91]** W. Chen. Declarative specification and evaluation of database updates. In C. Delobel, M. Kifer, and Y. Masunaga, editors, Deductive and Object-Oriented Databases (DOOD'91), number 566 in LNCS, pages 147-166, Munich, Germany, Dec. 1991.

[26] **[Chikayama 83]** Chikayama, T. "ESP-Extended Self-contained Prolog-as a preliminary kernel language of Fifth Generation computers. New Generation Computing, 1:11-24, 1983.

[27] **[Chikayama 84]** Chikayama, T. "Unique Features of ESP". Proc. Int'l Conf. on Fifth Gen. Comp. Sys. (1984), pp. 292-298.

[28] **[Clark 86]** K.L. Clark and Gregory, "PARLOG : A parallel Logic Programming Language", ACM Trans. on Language and Systems 8, 1 (january 1986), 1-49.

[29] **[Clark 87]** K.L. Clark and Gregory, "Parlog and Prolog United", Proc. of the 4th Int. Conf. on Logic Programming, Cambridge, Mass: MIT Press, pp. 927-961, 1987.

[30] **[Conery 87a]** John S. Conery. "Hoops : an object-oriented Prolog". Technical Report, University of Oregon, 1987.

[31] **[Conery 87b]** John S. Conery. "Object-Oriented programming with First-Order Predicate Calculus". Technical Report CIS-TR-87-09, University of Oregon, Aug. 1987.

[32] **[Conery 88a]** J. Conery. Logical Objects. Proc. of the Fifth Int'l Conf. on Logic Prog. , p.p. '20-443, 1988.

[33] **[Conery 88b]** John S. Conery. "Hoops - user's Manual." Technical Report CIS - TR - 88 - 12 , Dept. Computer and Information Science, University of Oregon, Eugene, Oregon 1988.

[34] **[Davison 88]** A. Davison. POLKA: a PARLOG object-oriented language. Technical report, DOC, Imperial College, London, 1988.

[35] **[Davison 89a]** A. Davison. A survey of logic programming-based object-oriented languages. Technical Report 92/3, University of Melbourne, Jan. 1992. 4th revision; first published April 1989.

[36] **[Davison 89b]** Andrew Davison. “Polka: A Parlog Object Oriented Language”. PhD thesis, Imperial College, Dept. of Computer Science, London, September 1989.

[37] **[Davison 91]** A. Davison. From PARLOG to POLKA in two easy steps. In J. Maluszynski and M. Wirsing, editors, Third International Symposium on Programming Language Implementation and Logic Programming, PLILP'91, number 528 in LNCS, pages 171-182. Springer-Verlag, 1991.

[38] **[Davison 92]** A. Davison. A survey of logic programming-based object-oriented languages. In Object-Based Concurrency (Wegner P., Yonezawa A. and Agha G., eds.) Reading, Mass.: Addison-Wesley.

[39] **[Davison 93]** Davison, A. "A Survey of Logic Programming-based Object Oriented Languages". In Research Directions in Concurrent Object-Oriented Programmaing. The MIT Press, Cambridge, MA, 1993.

[40] **[Doma 86]** Doma, A. "Object-Prolog: Dynamic Object-Oriented Representation of Knowledge". SzKi Comp. Research and Inn. Center (1986), 14 p.

[41] **[Doma 88]** 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.

[42] **[Fukunaga 86]** K. Fukunaga and S. Hirose. An experience with a Prolog-based object-oriented language. In N. Meyrowitz, editor, OOPSLA'86: Object-Oriented Programming Systems, Languages and Applications (OOPSLA’86): Conf. Proc., Portland, OR, USA, 29 Sep - 2 Oct 1986, pages 224-231. 1986. (SIGPLAN Notices, 21(11)).
[43] **[Furukawa 84]** K. Furukawa, A. Takeuchi, S. Kunifuji, H. Yasukawa, M. Ohki, and K. Ueda. "Mandala : A logic based knowledge programming system". In International Conference on Fifth Generation Computer Systems, Tokyo, Nov. 1984.
[44] **[Gallaire 86]** Gallaire, H. "Merging Objects and Logic Programming: Relational Semantics, Performance and Standarization". In Proc. AAAI'86, pp.754-758, Philadelphia, Pensylrania, 1986.
[45] **[Gandilhon 87]** Gandilhon T. “Proposition d’une extension objet minimale pour Prolog.”, Actes du séminaire de Programmation en Logique, Trégastel (mai 1987), pp. 483-506.
[46] **[Gandriau 88]** Gandriau, M. "CIEL: classes et instances en logique". Thèse de Doctorat, ENSEEIHT 1988, 151p.
[47] **[Girard 87]** J.-Y. Girard. Linear logic. Theoretical Comput. Sci., 50:1-102, 1987.
[48] **[Girard 89]** J.-Y. Girard. “Introduction à la logique linéaire”, in Logique et Informatique : une introduction, INRIA, B. Courcelle ed., Paris 1989.
[49] **[Gloess 84]** P.Y. Gloess, "Logis, un système Prolog dans un environnement Lisp". Actes du séminaire de programmation en logique. pages 213-222, Plestin les Grèves. Avril 1984.
[50] **[Gloess 86]** P.Y. Gloess, J. Marcovich. "OBLOGIS, a flexible implementation of Prolog logic and its application to the design of broaching expert system." First Int. Conf. on Applications of A.I. in Engineering problems. Southampton, pages 1-21. Avril 1986.
[51] **[Gloess 89a]** P.Y. Gloess, "Prolog et Objets et Objets et Prolog", Présentation groupe AFCET-PROLOG, Paris, Mai 1989.
[52] **[Gloess 89b]** Gloess, P.Y. "ULog, Aspect Formels et Pratiques d'un Interface entre Programmation Logique et Objets". Actes du 8è Séminaire de Programmation en Logique, pp. 71-96, Mai 1989.
[53] **[Gloess 90]** Gloess, P.Y. "Contribution à l'optimisation de mécanisme de raisonnement dans des structures spécialiées de représentation de connaissances". Thèse d'état, Univ. de TechnWorldLogie de Compiègne, Janv. 1990.
[54] **[Gloess 91]** Gloess, P.Y. “U-Log, A Unified Object Logic”, Revue d’Intelligence Artificielle, Vol. 5, n° 2/1991, pp. 33-66.
[55] **[Gloess 95]** Gloess, P.Y. M. Oros, C.M. LI, “U-Log3 = DataLog + Constraints”, (Prototype) Actes des JFPL95, Dijon (France), pp. 369-372.
[56] **[Goldberg 83]** Goldberg, A. and Robson, D. "Smalltalk-80 : The language and its implementation". Addison-Wesley, 1983.
[57] **[Grant 90]** J. Grant and T. K. Sellis. Extended database logic. complex objects and deduction. Information Sciences, 52(1):85-110, Oct. 1990.
[58] **[Herbrand 67]** J. Herbrand "Investigations in Proof Theory", in From Frege to Gödel : A Source Book in Mathematical Logic, 1879-1931, van Heijenoort, J. (ed.), Harvard University Press, Cambridge, Mass. 1967, 525-581.
[59] **[Ishikawa 86a]** Ishikawa, Y. et Tokoro, M. A Concurrent Object Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation, Actes de OOPSLA'86, ACM Sigplan Notices 21, 11 (Novembre 1986), pp. 232-241.
[60] **[Ishikawa 86b]** Y. Ishikawa and M. Tokoro. ORIENT84/K: A language with multiple paradigms in the object framework. In Nineteenth Annual Hawaii Int. Conference on System Sciences, volume II: Software Track, Honolulu, HI, Jan. 1986.
[61] **[Ishikawa 87]** Ishikawa, Y. et Tokoro, M. Orient84/K: An ObJect Orlented Concurrent Programming Language for Knowledge Representation, Object-Oriented Concurrent Programming (1987), W 159-198.
[62] **[Iwanaga 91]** R. Iwanaga and O. Nakazawa. Development of the object-oriented logic programming language CESP. Oki Technical Review, 58(142):39-44, Nov. 1991.
[63] **[Jungclaus 93]** R. Jungclaus. Logic-Based Modeling of Dynamic Object Systems. PhD thesis, Technical University Braunschweig, Germany, 1993.
[64] **[Kahn 86a]** K. M. Kahn. VULCAN: Logical concurrent objects. In E. S. Shapiro, editor, Concurrent PROLOG: Collected Papers, volume 2, pages 274-303. MIT Press, 1986.
[65] **[Kahn 86b]** Kahn, K., Tribble, E.D., Miller, M.S. & Bobrow, D.G. "Objects in Concurrent Logic Programming Languages". Actes de OOLPSA'86, ACM Sigplan Notices 21, 11 (1986), pp. 242-257.
[66] **[Kahn 87]** 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 [Shapiro 87a])
[67] **[Khoshafian 86]** S. Khoshafian, G. Copeland, “Object Identity”, Sigplan Notices, n° 21, 1986.

[68] **[Kifer 95**] M. Kifer, "Deductive and Object Data Languages : A Quest for Integration",Keynote address at the 4-th Intl. Conf. on Deductive and Object-Oriented Databases, Singapore, December 1995 (Springer's LNCS 1013).

[69] **[LPA 2017]** Prolog++ toolkit, an expressive and powerful object-oriented programming system, which combines the best of AI and OOPs. LPA 2017. http://www.lpa.co.uk/ppp.htm

[70] **[Lieberman 86]** H. Lieberman, "Using prototypical objects to implement shared behaviour in object oriented systems". In N. Meyrowitz ed. OOPSLA'86, pages 214-223. Portland, september 1986.

[71] **[Lindholm 87]** T.G. Lindholm, R.A. O'Keefe, "Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code", Actes 4th Int'l Conf. on Logic Prog., pp.21-39, 1987.

[72] **[Machanda 88]** Manchanda, S. & Warren, D.S. "A Logi-based Language for Database Updates". Actes W. on Foun. of Ded. Db. and Logic Programming, pp. 363-394, 1988.

[73] **[Malenfant 89a]** 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.

[74] **[Malenfant 89b]** 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, Nothingham, UK, July 1989.

[75] **[Malenfant 89c]** Malenfant, J., Lapalme, G. et Vaucher, J. ObjVProlog-D: Distributed Knowledge Processing using Concurrent Objects, note pour la table ronde sur OOCP, ECOOP'89 (juil. 1989),3 p.

[76] **[Malenfant 89d]** Malenfant, J., Lapalme, G. et Vaucher, J. Coherent State Changes for Logic Objects, soumis à J. of Logic Programming (août 1989).

[77] **[Malenfant 90a]** J. Malenfant, G. Lapalme, and J. Vaucher. Metaclasses for metaprogramming in logic. In Second International Symposium on Meta-Programming in Logic, pages 257-271, Leuven, Belgium, Apr. 1990.

[78] **[Malenfant 90b]** 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.

[79] **[Malenfant 91a]** J. Malenfant, G. Lapalme, and J. Vaucher. ObjVProlog-D: A reflexive object-oriented logic language for distributed computing. OOPS Messenger, 2(2):78-81, Apr. 1991.

[80] **[Malenfant 91b]** J. Malenfant, G. Lapalme, and J. Vaucher. Coherent state changes for logic programs. Research report LITP 91-01 RXF, Équipe mixte LITP/RXF, Jan. 1991.

[81] **[Malenfant 92]** Malenfant, J. "Architecture méta-réflexives en programmation logique par objets". JFPL 92, pp. 253-267, 1992.

[82] **[Mancarella 95]** 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.

[83] **[McCabe 92]** F. G. McCabe. Logic & Objects. International Series in Computer Science. Prentice-Hall, 1992.

[84] **[Meyer 87a]** Meyer B. "Eiffel : Programming for reusability and extendibility.", ACM SIGPLAN Notices, 22(2):85-94, 1987.

[85] **[Meyer 87b]** B. Meyer, "Reusability: The Case for object-oriented Design", IEEESoftware 4, 2 (Mars 1987), pp.50-64.

[86] **[Meyer 88]** B. Meyer. "Object-Oriented Softuare Construction". Prentice Hall, New York, 1988.

[87] **[Meyer 90]** Meyer B. "Conception et programmation par objets, pour le génie logiciel de qualité", InterEditions, Paris 1990.

[88] **[Misoguchi 84]** F. Misoguchi, H. Owhada, and Y. Katayama. "LOOKS: Knowledge representation system for designing expert systems in logic programming framework". In International Conference on Fifth Generation Computer Systems, ICOT, Japan, 6-9 Nov 1984, pages 606-12. North-Holland, Amsterdam, 1984.

[89] **[Miyoshi 84]** H. Miyoshi and K. Furukawa. Object-oriented parser in the logic programming language ESP. In Natural Language Understanding and Logic Programming, First International Workshop, pages 107119, Rennes, France, Sept. 1984. North-Holland.

[90] **[Moss 86]** Moss, C. CUT & PASTE - defining the Impure Primitives of Prolog, Third Int7 Conf. on Logic Prog. LNCS 225 (Juillet 1986), pp. 686-694.

[91] **[Moss 90]** Moss C.. An introduction to Prolog++. Research Report DOC 90/10, Imperial College, London, June 1990.

[92] **[Moss 94**] Moss C., "Prolog++ : The Power of Object-Oriented and Logic Programming", Addison-Wesley, 1994.

[93] **[Moteiro 89]** Monteiro, L. et Porto, A. Contextual Logic Prog., Actes 6th Int7 Conf. on Logic Prog., Portugal (1989), pp. 284-299.

[94] **[Ngomo 95a]** 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.

[95] **[Ngomo 95b]** 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.

[96] **[Ngomo 95c]** 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.

[97] **[Ngomo 96]** 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.

[98] **[Ohki 87]** M. Ohki, A. Takeuchi, and K. Furukawa. "An object-oriented programming language based on the parallel logic programming language KL1". In J.-L. Lassez, editor, Fourth International Conference on Logic Programming, MIT Press Series in Logic Programming, pages 894-909, 1987.

[99] **[Ohki 88]** M. Ohki, A. Takeuchi and K. Furukawa, "An object-oriented programming language based on the parallel logic programming language KL1". In Proc. FGCS'88, 895-909, Tokyo: ICOT.

[100] **[Shapiro 83a]** Shapiro, E., "A subset of Concurrent Prolog and its Interpreter", Tech. Report TR-003, ICOT-Institute for New Generation Computer Technology, Tokyo, Japan, January, 1983.

[101] **[Shapiro 83b]** Shapiro, E. and A. Takeuchi, "Object-Oriented Programming in Concurrent Prolog", New Generation Computing, 1:25-48, 1983. (Also Chap. 29 in [Shapiro 87]).

[102] **[Shapiro 86]** Shapiro, E. "Concurrent Prolog : A progress report". IEEE Computer 19, pp. 44-58, Aug. 1986. (Also Chap. 5 in

[103] **[Shapiro 87]** E. Shapiro, (Editor), "Concurrent Prolog", Vol. 1 and 2, MIT Press, 1987.

[104] **[Shapiro 89]** E. Shapiro, "The familly of Concurrent logic programming languages", Technical Report CS89-08, Depart. of Applied Mathematics and Computer Science, The Wietzmann Institute, Rehovot, 1989.

[105] **[SICStus Prolog,2017]** SICStus Prolog, state-of-the-art, ISO standard compliant, Prolog development system. https://sicstus.sics.se/

[106] **[Stabler 86]** E. P. Stabler, Jr. Object oriented programming in PROLOG. AI Ezpert, pages 46 57, Oct. 1986.

[107] **[Steele 90]** Steele G. L. "Common Lisp : the language" second edition, Digital Press, 1990.

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

[109] **[Stroustrup 92]** Stroustrup, B. "Le Langage C++". InterEditions 1992.

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

[111] **[Uustalu 91]** T.Uustalu, "Combination of Object-Oriented and Logic Paradigms", Master of Engineering Thesis, Tallinn Technical University, Written at the Dept. of Computer Systems and Telematics, Apr-Aug 1991.

[112] **[Uustalu 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.

[113] **[Vaucher 88]** J. Vaucher, G. Lapalme, and J. Malenfant. SCOOP: Structured concurrent object oriented Prolog. In S. Gjessing and K. Nygaard, editors, ECOOP'88: Europ. Conf. on Object Oriented Programming(ECOOP'88), Proc., Oslo, Norway, 15 17 Aug 1988, pages 191 211. Springer Verlag, Berlin, 1988. (LNCS, 322).

[114] **[Warren 84]** D. Warren. Database updates in pure PROLOG. In Fifth Generation Computer Systems, pages 244 253. ICOT, 1984.

[115] **[Zaniolo 84]** Zaniolo, C. "Object-Oriented Programming in Prolog". In Proc. of the IEEE Internatial Symposium on Logic Programming, pp. 265-270, Atlantic City, New Jersey, 1984.