In CICLOPS 2002 (ICLP Workshop), Copenhagen

The O'Ciao Approach to Object Oriented Logic Programming

Angel Pineda and Francisco Bueno

Technical University of Madrid (UPM), Spain
{apineda,bueno}@fi.upm.es

Abstract. There have been quite a number of proposals for the integration of Object Oriented Programming features into Logic Programming, resulting in much support theory and several languages. However, none of these proposals seem to have made it into the mainstream. Perhaps one of the reasons for this is that the resulting languages depart too much from the standard logic programming languages to entice the average Prolog programmer. Another reason may be that most of what can be done with object-oriented programming can already be done in Prolog through the meta- and higher-order programming facilities that the language includes, albeit sometimes in a more cumbersome way. In light of this, in this paper we propose an alternative solution which is driven by two main objectives. The first one is to explicitly include at the language level only those characteristics of object-oriented programming which are cumbersome to implement in standard Prolog systems. The second one is to do this in such a way that there is a minimum impact on the syntax and complexity of the language, i.e., to introduce the minimum number of new constructs and concepts. Finally, we would also like the implementation to be as straightforward as possible, ideally based on simple source to source expansions.

1 Introduction

Over more than two decades now the logic programming community has seen multiple proposals which try a combination of the object oriented and logic programming paradigms (OOP and LP). Shapiro and Takeuchi [21] are the pioneers in this matter with a model of objects as *perpetual processes* in a committed-choice language without backtracking. Object state is stored in unshared variables, and object inter-communication is achieved using shared variables. An implementation, based on KL1, of this idea can be found in [16].

Most proposals can be classified into several groups, based on the different design and implementation approaches, as suggested by A. Davison in [11]. The *committed choice view* is as mentioned above. In the *backtracking process view*, an object is also treated as a perpetual process, but backtracking is allowed. Object inter-communication is based on message-passing techniques other than variable sharing. See [14,10,19]. In the *clause view*, objects are treated as dynamically generated clauses which record the state. Object inter-communication is based on execution of goals (methods). [22] and [15] are excellent references in this line. The *meta-interpretation view* is based on the transformation of object-oriented code into the underlying logic language. [20], [18], [17], [9], [13], and [22] are examples of this technology. Another group may be added to this classification: Technologies based on *higher-order logic*. See [6,12,4].

In general, in all approaches the main challenge is *state manipulation*. V. Alexiev states this in [3] and provides another classification based on different solutions to this problem. Inheritance has also been widely discussed. Bugliesi [5] presents a declarative point of view on inheritance. LIFE [1] and LOGIN [2] also state similar interesting points of view. However, in most proposals a procedural view of inheritance (e.g., message forwarding and delegation) is taken.

We pursue a different approach to the subject. Our aim is to integrate object oriented behavior into a logic language as smoothly as possible. Consider the following: There are a few very simple concepts behind the OOP paradigm, and most of them can be emulated within Prolog. This allows us to offer an alternative point of view: We can reinterpret some Prolog characteristics so as to suit object-orientation. Thus, we take an existing LP platform as starting point. Then, we identify those concepts present in the language which are very similar to those involved in OOP. Finally, we adjust the language level to smoothly (both in expressiveness and in implementation) obtain the object oriented behavior.

Traditionally, four main concepts are involved in OOP: (1) State encapsulation (attributes within classes), (2) Instantiation (objects), (3) Inheritance, and (4) Polymorphism. State can be found in LP in the form of *dynamic* predicates. Whenever those dynamic predicates are constituted by simple facts they are a good representation of *changing* state. Encapsulation is also found in LP from *modules* (see, for example, the Ciao module system [7]), once state is represented as predicates. Polymorphism is also possible in LP simply because Prolog is not a typed language; or, to put it in other words, since Prolog is not typed, polymorphism is not an issue (unless we added types to LP). Object inter-communication is not an issue: Static predicates can be promoted to *methods* in the same way as procedures are in imperative languages such as C++ or Java.

There are only two key concepts of OOP which are not found at the roots of LP: *instantiation* and *inheritance*. These two will concentrate most of our attention in introducing O'Ciao, a library package implemented in the Ciao language in order to enable OOP (with a similar spirit to, e.g., SICStus Objects). First, we give a description of this library via examples (Section 2). Then we discuss in Section 3 some implementation issues, and in Section 4 present some experimental results, aimed at measuring the impact on performance of the OOP extension with respect to plain LP. We conclude in Section 5.

2 The O'Ciao Object Oriented Model

O'Ciao has been designed as an extension to the module system of the underlying LP language. In particular, we use the module system of the Ciao language. As a result, there is very little special syntax (other than syntactic sugar) for using the object-oriented features. This is another point of difference between O'Ciao and previous proposals. In order to do this, we have taken advantage of Ciao *packages*, which allow a uniform way of adding language features integrating compile-time expansions and run-time support (see [7]).

Classes are declared in O'Ciao much in the same way as modules, but adding Ciao package `class`, which will transform the original source code via a source-to-source expansion which is called the *class expansion.* At the conceptual level, there is a semantic "twist" of some declarations. Basically, dynamic predicates in a class become attributes, and exported predicates become public methods.

In order to import classes and use its objects in a module/program, the Ciao package `objects` has to be included. This will load a run-time support library, declare some syntactic sugar, and enable the operator for instance creation and a declaration which establishes class usage relationships (much the same as the `use_module/1` declaration establishes module usage relationships).

Class declaration. Classes are modules including the `class` package; attributes are declared with `dynamic/1`, public methods with `export/1`.

Example 1. The following Ciao code declares a class which implements an "imperative-style" stack of elements:

```
:- module(stack,[],[class]).  % or:  :- class(stack).
:- dynamic storage/1.     :- export(push/1).      :- export(pop/1).
push(Item) :- nonvar(Item), asserta(storage(Item)).
pop(Item) :- var(Item), retract(storage(Item)).
```

Notice the similarity with a traditional module. In fact, this kind of class may be called an *instantiable module*, just because there is no usage of object oriented specific declarations (e.g., inheritance relationships). The concept of instantiable module is a step prior to the concept of class. There is only one point of difference between a module and an instantiable module: "copies" of the instantiable module can be generated.

Object creation and manipulation. The `objects` package enables the `new/2` operator and the `use_class/1` declaration, which declares "imported" classes. The creation of objects (of the imported classes) is performed by the `new/2` operator. This operator takes a free variable as first argument (which will be bound to a unique instance identifier), and a class constructor as second argument. Instances can also be statically declared. Once the object has been created, any exported (public) predicate may be called as a classically module-qualified goal.

Example 2. The following code uses the class defined in the previous example:

```
:- use_class(stack).
:- stack1 instance_of stack.
main :- Stack2 new stack, stack1:push(a), Stack2:push(c).
```

In the example above, `stack1` is declared to be an *instance* of the stack module. This instance is used as if it was a loaded module (a "copy" of the stack module). Additionally, instances may be also created at run-time (this is the case of Stack2) by using the new/2 operator. This accounts for instantiation, and is the only main new operator added to the language. Its implementation is discussed in Section 3.

Constructors are implicitly declared by writing clauses of a predicate whose functor matches the class name (any arity is allowed). Destructors are also implicitly declared by writing clauses for a destructor/0 predicate.

Example 3. Constructors and destructors are mainly used to allocate/release system resources at the proper time. For example:

```
:- class(file_reader).
:- export(get_char/1).
:- dynamic handler/1.
file_reader(FileName) :-   % constructor
     open(FileName,read,Handler), assert(handler(Handler)).
destructor :- retract(handler(H)), close(H).
```

The default constructor (with arity zero, i.e., identical to the class name) is used if defined, in the absence of a different explicit constructor when calling new/2. Instance creation fails whenever the given constructor fails.

Constructors are useful to perform some initialization just after object creation takes place. *State initialization* is a particular case which is also provided in the traditional way: Writing clauses for the state (dynamic) predicate, i.e., the attribute.

The objects package also provides some run-time type checking primitives: interface/2, instance_of/2 (not to be confused with the declaration of the same name), and derived_from/2. The first one is related to interface inheritance, the second one to code inheritance, and the third one simply retrieves the class which derived the involved object. Similar primitives are also used at compile time in order to perform a limited analysis on objects usage. This gives a chance to detect semantic errors such as calling a non-public method at compile-time. For the cases where this is not possible, run-time checks are introduced (using the above primitives).

Inheritance. O'Ciao makes use of two different kinds of inheritance relationships: code inheritance and interface inheritance. Multiple code inheritance is not supported, but emulated via interface inheritance (much the same as in Java).

Code inheritance is very similar to the re-exportation of modules through the (ISO Prolog) reexport/1 declaration. The main differences are:

- A redefined predicate is implicitly overridden by the new definition (whereas it is a name clash with modules). We believe implicit overriding is more natural in OOP.
- The visibility of predicates can not be restricted: An inherited (but not public) predicate may be exported, but an inherited public predicate can not become private.

The last point ensures a uniform public interface along the inheritance line. It is related to the existence of two different relationships in OOP: inheritance and publication. Inheritance is related to a set of predicates which we call the *inheritable interface* while (re)exportation is related to a set of predicates called the exported or *public interface* (public and exported predicates are the same concept in the scope of OOP).

Predicates are private by default, so they must be explicitly made public by exporting them. Public predicates are then inheritable by default. An extra declaration is added in order to explicitly declare private predicates which are inheritable (the rest are "completely" private, or *protected*). The declaration is `inheritable/1`. The (unique) class from which a given class inherits is then declared with `inherit_class/1`.

Example 4. The following example illustrates the usage of the declarations for inheritance. The `item` class declares `set_value/1`, `get_value/1`, and `datum/1` to be inherited by descendant classes; in particular, they are inherited by the `tagged_item` class.

```
:- class(item).
:- export(set_value/1).
:- export(get_value/1).
:- inheritable(datum/1).
:- dynamic datum/1.
set_value(X) :-
        retractall(datum(_)),
        assert(datum(X)).
get_value(X) :- datum(X).
```

```
:- class(tagged_item).
:- inherit_class(item).
:- export(set_tag/1).
:- export(get_tag/1).
:- dynamic tag/1.
set_tag(X) :-
        retractall(tag(_)),
        assert(tag(X)).
get_tag(X) :- tag(X).
```

Interface inheritance forces the class inheriting the interface to implement such interface. The compiler must ensure that a particular source will export the same set of predicates as another one (and implement the same attributes). Interface inheritance has been implemented by adding a declaration `implements/1`, where its argument may be a proper class or an *interface-expanded* source. Interface-expanded sources are similar to the interfaces provided in the Java language: Only `export/1` and `dynamic/1` declarations are allowed.

Example 5. The code below to the left declares an interface and the class to the right inherits it. Therefore, it is forced to implement public predicates `a/1` and `b/1`, in addition to those exported by `item`: `set_value/1` and `get_value/1`.

```
:- interface(is_a_must).
:- export(a/1).
:- export(b/1).
```

```
:- class(itf_example).
:- implements(is_a_must).
:- implements(item).
```

Method overriding. Overridden predicates in OOP are imported predicates which are locally redefined. A predicate is said to be overridden when it has been inherited from any ascendant class, but the same predicate has been defined at the current source class. In order to distinguish between both definitions there is an `inherited/1` predicate qualifier. By default, the local predicate is used. If the inherited predicate definition is to be called, it must be qualified as `inherited`.

Virtual methods. Virtual methods allow descendant classes to provide different implementations for a predicate. The ancestor will always call the version that is defined at the bottom-most successor class in the inheritance line. O'Ciao provides the `virtual/1` declaration in order to declare virtual methods.

Example 6. In this example, any object derived from class `integer_item` will accept (only) an integer as argument to `set/1`, regardless of the `validate_item/1` check implemented in the parent class `generic_item`.

```
:- class(generic_item).            :- class(integer_item).
:- virtual validate_item/1.        :- inherit_class(generic_item).
validate_item(I) :- nonvar(I).     validate_item(I) :- integer(I).
:- export(set/1).
set(X) :- validate_item(X), set_value(X).
```

Object self reference. Sometimes, an object needs to know a reference to itself. O'Ciao provides this feature through the `self/1` predicate.

3 Implementation issues

Providing instances requires, in essence, that the code be aware of the current instance being executing. In O'Ciao, this is achieved by extending the naming convention for module qualification to include object names, and expanding methods with an extra argument to pass around the object identifier.

In the module system, expressions of the form `M:goal` are qualified by module name `M` (an atom). In the object system, `M` will instead be an *object name*, which is a *unary term* whose functor matches the class which derives the object, and whose argument is the *instance identifier*, a unique atom for that instance. The `new/2` operator returns the object name of the created object. Thus, during execution, messages will have the form $class(objid) : method$.

When methods are called, the code involved will pass around the object name, so that the assert/retract predicates on the attributes operate over the proper state encapsulation. This is achieved by an extra argument, also known as the *hidden argument* in other object oriented languages. This argument is added by the class expansion to every clause of a static predicate of a class (dynamic predicates, i.e., attributes, are treated in a different way). The extra argument is added as the last argument in order to preserve the original Prolog indexing.

Therefore, messages of the form $class(objid) : method$ must be converted to goals of the form $class : method(objid)$. The Ciao module expansion facilities are used for this. In Ciao, meta-goals and meta-arguments of the form `M:Term` are expanded at run-time when `M` cannot be resolved at compile-time, which is the case with objects. The compiler sets things up so that a meta-goal `M:Term` in a module Source will be expanded by exp_goal(Term,Source,M,Goal) to Goal, and meta-arguments by a similar predicate exp_fact/4. These predicates are generated by the class expansion, and are discussed in the rest of this section.

The general system performance is fully dependent on the code automatically generated by the class expansion. In general, the more information is known by the class expansion, the better performance is achieved. The Ciao compiler allows code expansion in two stages (for further reference, consult [8]). In a first stage, declarations of a class are processed in order to generate the interface information that may be needed. In a second stage, module dependences have been resolved by the compiler. Code can then be generated taking into account other module or class declarations related to the current code being expanded. This interesting feature allows O'Ciao class expansion to avoid many run-time checks, to generate much static code, and to perform exhaustive semantic analysis on class code.

Promoting static predicates to methods. Besides adding the hidden argument to methods, the class expansion automatically generates clauses for exp_goal/4 to account for the run-time expansions of method calls. The code automatically generated also performs visibility checks, so that only classes imported can be used. Prolog indexing is preserved all along the expansion process.

Example 7. Consider a module m which imports class tagged_item of Example 4. The following code will be automatically generated:

```
accessible_set_value_1(item,item).
accessible_set_value_1(item,tagged_item).
accessible_set_value_1(item,m).
accessible_set_tag_1(tagged_item,tagged_item).
accessible_set_tag_1(tagged_item,m).
exp_goal(set_value(X),M,item(Obj),'item:set_value'(X,Obj)) :-
   accessible_set_value_1(item,M).
exp_goal(set_tag(X),M,tagged_item(Obj),'tagged_item:set_tag'(X,Obj)) :-
   accessible_set_tag_1(tagged_item,M).
```

Virtual methods require a different run-time expansion, because it is not possible to figure out which version of the predicate will be the most specialized one. This depends on the actual object calling the method. Therefore, it can only be determined at run-time. This is done by a run-time support module virtual_rt which includes the required inheritance information to determine the most specialized version of a virtual method.

Example 8. Method set/1 of Example 6 calls validate_item/1, which is a virtual method. This call will be expanded at compile-time as follows:

```
set(X,Obj) :- virtual_rt(Obj):validate_item(X), ...
```

Note that the object name itself does not contribute in any way to expand messages of the form Obj:goal into proper goals: It is just passed along as an argument. Thus, it is possible to call methods using compile-time generated clauses which avoid most of the run-time expansion overhead. This optimization can be applied to any method. Any goal of the form `Obj:goal` will be expanded at compile-time into `method_call(Obj,goal)`, where `method_call/2` is defined by (automatically generated) clauses of the form:

```
method_call(item(ObjId),Method) :- item_call(Method,ObjId).
item_call(set_value(X),ObjId) :- set_value(X,ObjId).
```

Promoting dynamic predicates to attributes. For attributes a different expansion scheme has been implemented, since adding an extra argument for the object name will most probably compromise efficiency when the number of instances of a class is high. Instead, different predicate names for the attributes of each instance are dynamically created. This is performed by run-time expansion `exp_fact/4` and operator `new/2`.

However, the expansion is different for external (i.e., from outside the class) and internal manipulation of the attribute. Whenever an external call, assert, or retract is issued on an exported attribute, the code is run-time expanded, as it is done with method calls, but in this case using `exp_fact/4`.

Example 9. For an exported attribute `datum/1` of a class `export_item` the following clause will be generated at compile-time:

```
exp_fact(datum(X),M,export_item(Obj),Goal) :-
   accessible_datum_1(export_item,M),
   functor_concat(Obj,':export_item::datum'(X),Goal).
```

where the auxiliary predicate `functor_concat/3` has computed answers of the form `functor_concat(class(123),pattern(X),'123pattern'(X))`.

Whenever a method operates over an attribute (i.e., internal attribute manipulation), a different expansion is performed. A run-time support module `class_rt` is used to map the attribute to the corresponding dynamic predicate. An optimization is now possible by using special versions of the assert/retract predicates. Most times, assert/retract predicate calls may be *automatically* translated to their specialized versions at compile time, saving some overhead.

Example 10. Method `set_value/1` of class `item` in Example 4 will be expanded at compile-time to:

```
set_value(X,Obj) :- ..., assert_attr(Obj,':item::datum'(X)).
```

When a new object is created, the dynamic predicates which hold its state must also be (dynamically) created. This includes all attributes along the complete inheritance line of the class which derives the object. The names of attributes are saved at compile-time in *attribute templates*, so that `new/2` uses the templates to create the corresponding dynamic predicates for the instance.

Example 11. Attribute `datum/1` of class `generic_item` of Example 6 will be assigned a template of the form:

```
attribute(generic_item,':generic_item::datum',1).
```

4 Performance tests

During the development of O'Ciao, we have tested several implementation alternatives before reaching the one described in this paper. Those prototypes have shown very helpful in determining which might be the main weaknesses in terms of performance and how to solve them. Not surprisingly, the main points to take into account are instance creation/destruction, attribute manipulation (i.e., usage of assert/retract on attributes), and method calling. In this section we present the results of the performance benchmarking conducted on these issues for comparing the OOP extension with plain LP.

Instance creation and destruction. Instance creation/destruction may be considered as "wasted time" since no user code is executed (except for constructors and destructors). There are two main factors which influence the performance of instantiation: the number of attributes needed by the object and the number of previously created (and not destroyed) objects.

Since creating an object basically amounts to creating its attributes, the number of attributes is an obvious factor. Figure 1 shows the relationship between the number of attributes and the execution time for creating and immediately destroying a single object. The results indicate a less-than-linear,[1] quite acceptable, overhead.

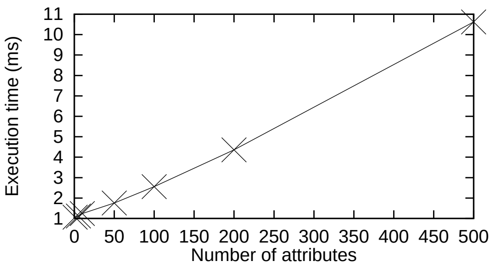

Fig. 1. Creating and destroying one object

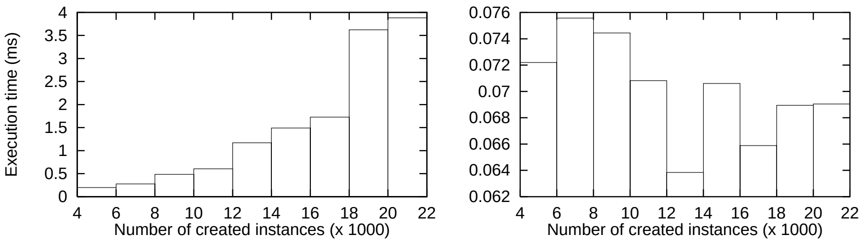

Fig. 2. Creating (versus destroying) objects

[1] Don't let the scale confuse you: It is less than linear because, e.g., for 500 attributes it takes less than ten times the time for 50 attributes.

Every time an object is created, new atoms for the instance identifier and the attribute names are created. Thus, the atom table itself grows up. Unfortunately, every time an atom is created the atom table must be looked-up first. As a result, object creation might perform worse with the number of instances created during execution. In the test shown in Figure 2, left, objects were not destroyed after creation. The overhead growth tends to be exponential with the number of instances. In some cases, the Prolog engine may run out of memory and temporarily stop execution in order to reserve additional space (this effect can be seen at the 18 Kobjects point). Fortunately, instance destruction alleviates this behavior completely, as it can be observed in Figure 2, right, where objects were destroyed *immediately* after creation. Notice that execution time is now kept at a minimum even when creating up to 22000 objects.

Attribute manipulation. To perform this benchmarking, we have compared the execution times of assert/retract when called from a module against when called from a class. Exactly the same code (except for the obvious syntactic differences) was used both in the module and the class. The results showed that the class code is 50% slower than the module code. This was due to the poor efficiency of the Prolog-coded predicate `functor_concat/3` used in the creation of names of attributes. A C-coded implementation of this predicate is expected to improve the performance of attribute manipulation.

Method calling. This test was mainly designed in order to measure the execution overhead involved in run-time code expansion when a method is called. Since an absolute measure will not be useful in order to reach a conclusion, we have compared execution times of calling methods against the execution of traditional module-qualified goals. Three kinds of goals have been considered:

- Dynamic[2] goals of the form `Var:goal`. These goals may be found both with the module system (when `Var` is bound to a module name) and with the object oriented extension (when `Var` is bound to an object identifier).
- Optimized dynamic goals: found in the object oriented extension, when `Var:goal` is translated to `method_call(Var,goal)` (as described in page 8).
- Static goals of the form `module:goal`, where `module` is known at compile-time. These are typically found with the module system, but also with the object oriented extension when objects are statically declared.

Table 1 summarizes the results on execution times of the abovementioned kinds of goals. Each of the two main columns shows execution times for one encapsulation system. The column labeled as *Ratio* is the quotient of the other two, indicating how much faster or slower (1/x) objects are than modules. The first two rows show that O'Ciao exhibits better performance on run-time expanded goals. This is just because object-oriented goals need simpler checks on the import/export interface. This result is very important, since dynamic goals are the usual method calling in OOP. The last two rows compare performance on static goals. This shows that static instances are rather expensive.

[2] "Dynamic" denotes here that the call can not be fully resolved at compile time.

Module system		O'Ciao		
Kind of goal	Time (ms)	Kind of goal	Time (ms)	Ratio
X:goal	0.0939	X:goal	0.01205	7.8
X:goal	0.0939	method_call(X,goal)	0.002155	43.6
mod:goal	0.0003627	method_call(X,goal)	0.002155	1/5.9
mod:goal	0.0003627	obj:goal	0.000457	1/1.3

Table 1. Goal execution times

Overall, the tests show that instance creation and method calling have an insignificant overhead on program execution. Considering that dynamic goals is the usual way of method calling, classes perform even better than modules. Attribute manipulation adds an important overhead, but we expect to alleviate it by careful implementation of the overhead sources.

5 Conclusions

We have focused the design and the implementation of O'Ciao on the final user requirements instead of the theoretical aspects. In this line, we expect O'Ciao to be an easy-to-learn programming tool both for programmers familiarized with Prolog and/or object-orientation. In O'Ciao we have avoided as much as possible the development of a new language, simply by introducing a minimal set of new features and keeping the original language syntax as much as possible. The implementation has been done exclusively using the available Prolog compiler, avoiding the development of a new one. Performance of the resulting object-oriented programs has also been considered. The results show that performance is quite acceptable, compared to plain Prolog.

We expect to further enhance current performance results simply by adding a few C-coded support primitives to our engine. Implicit object destruction is another target for future work related to implementation. Currently, O'Ciao instance destruction is explicitly invoked by calling a `destroy/1` operator. This operator can be seen as a handle for retracting the state of the object. Note that in Prolog dialects, retraction is the only way to clean up the predicate database. However, we are currently working on enhancing the Prolog engine garbage collector to do the work. The idea is to enable a hook predicate that would be called by the garbage collector on attributed variables. Dynamic predicates would have an attributed variable associated so that when it is "garbage collected" the retraction is done. This is possible if dynamic predicates are encapsulated, which is the case in the module system of Ciao and in objects in O'Ciao. Having this, it would then suffice to associate an attributed variable to each instance, and a reference count to this variable: this will enable implicit instance destruction.

Finally, we have not commented on the possible applications of O'Ciao, since it inherits much of the OOP application field. However, we have found it a very good tool in order to develop the Ciao/Java and Ciao/TclTk programming interfaces. Distributed and agent programming is another field of experimentation were O'Ciao is currently being applied.

References

1. Hassan Aït-Kaci. LOGIN: A Logic Programming Language With Built-In Inheritance. Tech. Rep., Microelectronics and Computer Technology Corporation, 1985.
2. Hassan Aït-Kaci. An Introduction to LIFE – Programming with Logic, Inheritance, Functions and Equations. In *ILPS*, 1993.
3. Vladimir Alexiev. Mutable Object State for Object-Oriented Logic Programming: A Survey. Tech. Rep., Dept. of Computing Science, University of Alberta, 1993.
4. Jean-Marc Andreoli and Remo Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. In *Proc. ICLP*, June 1990.
5. M. Bugliesi. A Declarative View of Inheritance in Logic Programming. In *Proc. of the Joint International Conference and Symposium on Logic Programming*, 1992.
6. Michele Bugliesi, Giorgio Delzanno, Luigi Liquori, and Maurizio Martelli. A linear Logic Calculus of Objects. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, September 1996.
7. D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *CL2000*, LNAI, 1861, pages 131–148. Springer-Verlag, July 2000.
8. D. Cabeza and M. Hermenegildo. The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
9. John S. Conery. Logical Objects. In *Proceedings of the Fifth International Conference and Simposium on Logic Programming*, August 1988.
10. Andrew Davison. *Polka : A Parlog Object Oriented Language*. PhD thesis, Imperial College, Dept. of Computer Science, 1989.
11. Andrew Davison. A Survey of Logic Programming-based Object Oriented Languages. Technical Report, University Of Melbourne, 1992.
12. Giorgio Delzanno and Maurizio Martelli. Objects in Forum. In *Proceedings of the 1995 International Symposium on Logic Programming*, December 1995.
13. Koichi Fukunaga and Shin ishi Hirose. An experience with a Prolog-based object-oriented language. In *Proc. OOPSLA '86*, 1986.
14. Man lai Tse, Wing hang Wong, and Ho fung Leung. P & P: A Combined Parlog and Prolog Concurrent Object-Oriented Logic Programming Language. In *Proceedings of the 1995 International Symposium on Logic Programming*, December 1995.
15. F. G. McCabe. Logic And Objects. Technical Report, Department of Computing, Imperial College, 1988.
16. M. Ohki, A. Takeuchi, and K. Furukawa. An Object-Oriented Programming Language based on the Parallel Logic Programming Language KL1. In *Proceedings of the Fourth International Conference on Logic Programming*, May 1987.
17. Z. Palaskas, P. Loucopoulos, and F. van Asshe. AMORE – object oriented extensions to Prolog. In *TOOLS '89*, 1989.
18. Ernesto Pimentel. L2||O2: A concurrent Object-Oriented Logic Language. In *Proceedings of the 1993 International Symposium on Logic Programming*, 1993.
19. Antoine Rizk, Jean-Marc Fellous, and Michel Tueni. An object-oriented model in the concurrent logic programming language Parlog. Tech. Rep., INRIA, 1989.
20. P. Schachte and G. Saab. Efficient Object Oriented Programming In Prolog. In *Logic Programming: Formal Methods and Practical Applications*, pages 205-243.
21. E. Shapiro and A. Takeuchi. *Object Oriented Programming in Concurrent Prolog*, pages 25–48. New Generation Computing, 1983.
22. Carlo Zaniolo. Object-oriented programming in Prolog. In *IEEE Symposium on Logic Programming*, 1984.