ObjVProlog: Metaclasses in Logic

Jacques Malenfant, Guy Lapalme and Jean Vaucher

Laboratoire INCOGNITO
Département d'informatique et de recherche opérationnelle, Université de Montréal
C.P. 6128, Succursale A, Montréal, Québec, Canada H3C 3J7

ABSTRACT. Scoop demonstrated the interest of class-based object-oriented logic programming languages [VAUC88] but it uses a simple model with no metaclasses. ObjVProlog is a new language based on the ObjVLisp model [COIN87] which provides metaclasses and unifies metaclass/class/instance concepts in a systematic and reflexive way. The ObjVProlog system and its implementation are described. Metaclasses are shown to be a very powerful tool. In this paper, we use them in an object-oriented way to develop a part-whole hierarchy and also for logic programming to define objects that behave like expert systems. Several other potential uses are discussed: faster objects, parallelism, object versions with virtual copy mechanisms and objects with a specialized logic.

KEWYORDS. ObjVProlog, logic programming, object-oriented programming, ObjVLisp, metaclasses, meta-programming, part-whole hierarchy.

1. INTRODUCTION

This paper presents an extension of logic programming towards object-oriented programming through the adaptation of the ObjVLisp model [COIN87] to Prolog. The resulting system is called ObjVProlog. This system illustrates the use of classes and metaclasses in structuring logic programs. Classes and metaclasses also give a powerful abstraction mechanism to create objects with particular behaviors. For example, they are used to define composite objects [BLAK87] and objects that behave like expert systems. Other potential applications are discussed.

Scoop and other such languages have demonstrated the benefits of object-oriented logic programming [VAUC88]. Scoop is a class-based language built around a block-structured syntax and providing parallelism. However, the class/instance model of Scoop is limited. Classes are statically defined and they are not objects. Scoop does not have metaclasses; all classes have the same pre-defined behavior.

ObjVProlog uses the ObjVLisp model to overcome these deficiencies. Parallelism is not addressed now but it will be discussed later (see Applications of metaclasses below). Two important improvements are making classes "first class citizens" [COIN87] and adding the metaclass concept. Metaclasses turn out to be a very powerful abstraction mechanism. This fact was already known in object-oriented programming; this paper shows that they can be very useful for meta-programming in logic.

In this paper, we first explore the problematic of combining logic programming and the object paradigm. Next, we briefly describe the ObjVLisp model and show the transition to ObjVProlog. A small planning system is given as example; it features metaclasses to define composite objects and expert system behavior. Future extensions based on metaclasses are discussed and related works are compared to ObjVProlog. ObjVProlog is interpreter-based, written in Quintus Prolog and running on Sun workstations. A more comprehensive version of this paper and complete code is also available [MALE89].

2. OBJECT-ORIENTED LOGIC PROGRAMMING

ObjVProlog proceeds by extension of a programming paradigm, logic programming, towards a new one, object-oriented programming [LEON88]. This extension is not straightforward because concepts do not transpose exactly from one paradigm to another.

2.1 Metaclass, class and instance

Presently, a clear notion of class in logic programming is emerging around the idea of an axiomatic, a theory or simply a "world" [LEON88]. A "world" is defined as knowledge in a domain of discourse expressed as sentences in a particular logic. In our work, a class is a description of the domain of discourse of its instances and, eventually, of the knowledge common to all its instances.

2.2 Methods and Instance variables

Methods in object-oriented programming map naturally to clauses: predicates become selectors and clauses represent the code. Instance variables or, from a more theoretic standpoint, the concept of modifiable internal state for objects poses serious problems. "Variables in logic programming behave differently from variables in conventional programming languages. They stand for an unspecified but single entity, rather than for a store location in memory" [STER86]. In order to realize state in object-oriented logic programming, two trends confront each other: procedural and declarative [LEON88].

ESP attributes [CHIK84] exemplify the procedural approach, being essentially imperative variables with traditional assignment. In Prolog, this can be implemented through assert-retract. The declarative approach may be implemented through actor-like behavior which calls for the creation of a new object with appropriate values upon a request for state modification [SHAP83]. An alternative approach uses Prolog-like facts and clauses to implement the declarative approach but relies on "assume-forget" predicates to give a cleaner semantics than "assert-retract". Tarau and Boyer illustrate this by what they call soft databases [TARA88]. We follow a similar approach making the effects of assume and forget

undone on backtracking, implementing a form of hypothetical reasonning. However, the semantics of a state varying logic is still a research theme and goes beyond the limits of this paper.

2.3 Message sending

Message sending may also take on a new face when introduced into logic programming. As said earlier, "logic" methods are implemented using Prolog-like clauses so that message sending can be viewed as a request for goal resolution using an object's clauses. The call-return semantics of message sending in traditional object-oriented language is thus traded for the success-failure semantics of logic programming [GALL86].

2.4 Multiple inheritance

Inheritance poses two problems: non-monotonicity, introduced by redefinition of methods among superclasses of a class, and ambiguity, caused by multiple inheritance which allows many definitions of a method to be visible when trying to answer a message. Many approaches have been proposed to tackle those issues. A monotonic semantics of inheritance can be given by using backtracking to consider every clause applying to a call [GALL86]. However, non-monotonic semantics is fundamental in object-oriented programming. Subclassing is used to build on super- classes but also to specify or distinguish behaviors. Non-monotonic behavior can be obtained through a "cut_inheritance" [GAND87] but this forces the use of a non-logical feature. Our approach employs non-monotonic inheritance and explicit use of "super" to access methods defined by superclasses.

Ambiguity of multiple inheritance is a well known problem in the object community. After much research and long discussions, the solution which tends to emerge now is defined by CLOS (Common Lisp Object System)[BOBR88]. The heart of the solution is an algorithm to compute linear extensions of classes, a total ordering on the graph of their superclasses. Touretsky suggests a partial order that may be nearer to the inheritance semantics [TOUR86] but we choose the CLOS approach here.

3. FROM OBJVLISP TO OBJVPROLOG

ObjVLisp, the model for our proposal, is characterized by its clean yet complete reflexive definition.

3.1 The ObjVLisp model

Metaclasses, classes and instances are fundamental concepts of object-oriented programming but they are not implemented the same way in every language. A uniform model was clearly needed. ObjVLisp is such a model [COIN87]. ObjVLisp proposes a model, minimum in a sense, where each concept is integrated in a systematic way. In ObjVLisp, metaclasses, classes and instances are all objects and an object is simply a collection of attributes or instance variables.

A class is a special kind of object that has certain required fields for its name, its superclasses' list, the list of its instances' variables and the list of its methods. There is no

structural differences between instances and classes. One can distinguish classes from instances because classes understand the message new for instance creation. A metaclass is simply a class that instantiates other classes. Inheritance in ObjVLisp is static for instances variables and dynamic for methods. Multiple inheritance is also provided.

The ObjVLisp model relies on two classes: `Class` and `Object`. `Object` is the class that defines the behavior of all objects in the system. `Class` is the class that defines the behavior of all classes and is the first metaclass. Every object in the system inherits from `Object` the instance variable "isit" which holds the name of its instantiation class. A class, being an object, also has the instance variable "isit" which holds the name of its metaclass. The other instance variables of classes are defined by the class `Class`: "name", "supers", "i_v" and "methods".

The creation of these two kernel classes uses a reflexive technique making both `Object` and `Class` transparent to the users [COIN87]. `Class` is the first object of the system and it is an instance of itself. This fact is made visible by the instance variable "i_v" of `Class` which is exactly the list of its own instance variables. First, `Class` is manually created and then, self-instantiated. Finally, the class `Object` is instantiated from `Class`.

`Object` does not have any superclass. It is the root of the *inheritance* graph. `Class`, being instance of itself, is the root of the *instantiation* graph. From these two classes, the programmer can create his own classes and instances by sending appropriate new messages to `Class` and then, to its newly created classes.

3.2 ObjVProlog

ObjVProlog adapts the concepts of ObjVLisp to the world of Prolog taking into account the problematic of section 2. The conceptual and implementation choices of ObjVProlog are the following:

1) an object in ObjVProlog is a Horn clauses database encoding the knowledge of the object,
2) classes (and metaclasses) are also objects,
3) clauses directly implement the state of objects which is modifiable through assume forget predicates,
4) inheritance has a non-monotonic semantics and relies on "super" to access superclasses' shadowed definitions and
5) multiple inheritance is provided using CLOS-like linear extensions.

The object database can only consist of predicates declared by the object's instantiation class. Both methods and instance variables are implemented as clauses and facts. Furthermore, ObjVProlog (like Scoop) distinguishes dynamic and static predicates: dynamic predicates can be modified during the life of the object while static predicates cannot. Inheritance in ObjVProlog is dynamic for both dynamic and static predicates. The object database is divided in levels each of which corresponding to the instantiation class of the object or a superclass.

Introductory example

To illustrate ObjVProlog, we present a simple example adapted from [COIN87]. We define the class `point` of pairs (x, y) as a subclass of `object` by instantiating the first metaclass `class`. Classes are created dynamically. The syntax uses '::' as message sending operator, the method new of `class` unifies its first argument to the name of the newly created class and its second argument is a list giving lists of clauses that will be used to initialize the dynamic predicates of the new class:

```
class::new(point,
   [[name(point)],
    [supers([object])],
    [dynamics([x/1, y/1])],
    [statics([
       [(replace_x(NX) :- ...)],
       [(replace_y(NY) :- ...)],
       [(initialize(...) :- ...)],
       [(display :- ...)]
     ])]]).
```

The class `point` is then used to create an instance with value (10, 20) and we send to it the message display (the output of the system is given in boldface characters):

```
point::new(P2, [[x(10)], [y(20)]]), P2::display.
10-20
```

Implementation

As in ObjVLisp, the kernel of ObjVProlog relies on two classes: `class` and `object`. The creation of the model proceeds by a manual creation of class followed by self-instantiation. The syntax used here consists of a pair (<object name>, <object database>). The object database is itself a list of pairs (<class name>, <dynamic predicates>) and each dynamic predicate is in turn represented as a pair (functor/arity, <list of clauses>). `dynamics/1` and `statics/1` have list of predicates as argument and use the same syntax recursively. So, the initial database of `class` is:

```
(class, [(class,
  [(isit/1,              [isit(class)]),
   (name/1,              [name(class)]),
   (supers/1,            [supers([])]),
   (linear_extension/1,  [linear_extension([class])]),
   (dynamics/1,          [dynamics([(isit/1,[]), (name/1,[]),
                                    (supers/1,[]),
                                    (linear_extension/1, []),
                                    (dynamics/1,[]),
(statics/1,[])])]),
   (statics/1,           [statics( [
      (new/2,
         [(new(Object, Clauses) :- ...)]),
      (basicnew/2,
         [(basicnew(Object, Class) :- ...)]),
      (initialize/3,
         [(initialize([], _, [])),
          (initialize([Class | Cs], Object, C) :- ...)]),
```

```
        (is_dynamic/1,
           [(is_dynamic(F/N) :- ...)])
      ])])])]).
```

From this manual definition we proceed to the instantiation of `class` and `object`. After two intermediate instantiations needed to implement correctly the dynamic inheritance of predicates [MALE89], the final forms of `object` and `class` are:

```
class::new(object,
   [[name(object)],
    [supers([])],
    [dynamics([isit/1])],
    [statics([
       [(class(C) :- ...)],          % instantiation class
       [(initialize([], _, [])),     % initialization at creation
        (initialize([Class | Cs], Object, C) :- ...)],
       [(is_class :- ...)],          % true if the object is a class
       [(is_metaclass :- ...)],      % true if the object is a metaclass
                                     % Class declare F/N as dynamic
                                     % predicate
       [(dyn_pred_s_class(F/N, Class) :- ...)],
       [(dyn_pred_s_class([C | Cs], F/N, Class) :- ...)],
       [(assumez(C) :- ...)],        % assume clause C as last clause
       [(assumea(C) :- ...)],        % assume clause C as first clause
       [(forget(C) :- ...)]          % forget the first clause unifiable
      ])])).                         % to C

class::new(class,
   [[name(class)],
    [supers([object])],
    [dynamics([name/1, supers/1, linear_extension/1,
               dynamics/1, statics/1])],
    [statics([
       [(new(Object, Clauses) :- ...)],
       [(basicnew([], _)),           % allocate and shape space
        (basicnew([Class | Cs], Object) :- ...)],
       [(understands(F/N) :- ...)], % do class or supers understand
       [(understands([C | Cs], F/N) :- ...)],                  % F/N
       [(is_static(F/N) :- ...)],    % do I declare F/N as static
       [(is_dynamic(F/N) :- ...)]    % do I declare F/N as dynamic
      ])])]).
```

This completes the ObjVProlog kernel. The ObjVProlog interpreter defines the ObjVProlog database (that contains all objects' databases), ObjVProlog predicates (to manipulate the database and some convenient ones), and executes ObjVProlog code. Essentially, it manages the context of execution (an object at a certain class level), context switching when a message is sent and reduces goals as in any Prolog meta-interpreter.

4. EXAMPLE

In this section we give the example of a simple assembly line planning system which illustrates the power of metaclasses and classes in ObjVProlog. Metaclasses and classes will be used to implement a part-whole hierarchy and objects that replace the standard Prolog

interpreter by an expert system inference engine.

The object that models the assembly line is a composite object and its parts are the three machines of the line. This object is also an expert system object and it uses this behavior to plan the sequence of machines to be used in processing, given a sequence of operations. We first present the classes and metaclasses to implement the part-whole hierarchy and the expert system behavior. After, we define the class modelling machines and the assembly line.

4.1 Part-whole hierarchies

In many applications, the concept of part-whole hierarchy is of great importance. However, a part-whole hierarchy is not properly modeled by the traditional inheritance hierarchy, as demonstrated by Blake and Cook [BLAK87]. Following [COIN88], we propose an implementation of a part-whole hierarchy based on two new classes: `part_whole_class` and `part_whole_object`. `part_whole_class` is the metaclass that allows the creation of classes of a part-whole hierarchy. `part_whole_object` is the equivalent of `object` in the inheritance hierarchy and defines the protocols to send messages to parts of an object. `part_whole_class` is simply a subclass of `class` that adds the dynamic predicate "parts", which holds the list of parts of an instance as pairs <part name>:<class>, and defines static predicates to create a part-whole object by recursively creating subparts:

```
class::new(part_whole_class, [
   [name(part_whole_class)],
   [supers([class])],
   [dynamics([parts/1])],
   [statics([
      [(new(Object, Clauses) :- ...)], % Create a new composite object
      [(create_parts(...)),            % Create parts of the object
       ...],                           % recursively
      [(part_understands(F/N) :- ...)] % Does a part understand F/N
     ])]
  ]).
```

`part_whole_object` defines three protocols to send messages to parts of an object. to_a_part(P:M) sends the message M to the part named P. to_part(M) sends the message M to the first part (in a depth-first order) that understands M (and to other parts by backtracking). to_all_parts(M) sends the message M to all parts that understands it in a failure-driven loop:

```
class::new(part_whole_object, [
   [name(part_whole_object)],
   [supers([object])],
   [dynamics([])],
   [statics([
      [(to_a_part(Part:Message) :- ...)],
      [(to_part(Message) :- ...)],
      [(to_all_parts(Message) :- ...),
       ...]
     ])]
  ]).
```

4.2 Expert systems

`expert_system_object` defines a simple meta-interpreter acting as an expert system taken from Sterling and Shapiro [STER86]. This meta-interpreter solves goals by backward chaining and generates an explanation of the proof at the end of a query. `expert_system_class` is the class from which expert system objects will be instantiated and it says that such objects have rules, askable predicates, callable predicates (solved directly by ObjVProlog) and allows memorization of user answers through known and untrue facts:

```
class::new(expert_system_object, [
   [name(expert_system_object)],
   [supers([object])],
   [dynamics([])],
   [statics([
      [(solve(Goal) :-          % Expert system inference engine
         solve(Goal, Proof),    % solve the goal building a proof tree
         interpret(Proof),      % Interpret the proof tree to give
         ...)],                 % the user an explanation of the proof
      [(solve(..., ...)),
       ...],
      [(interpret(...) :- ...),
       ...],
      ...
     ])]
  ]).

class::new(expert_system_class, [
   [name(expert_system_class)],
   [supers([expert_system_object])],
   [dynamics([rule/2, callable/1, askable/1, untrue/1, known/1])],
   [statics([])]
  ]).
```

4.3 The simple planning system

The assembly line object will be made of machine objects having capability in terms of operations that they can perform:

```
class::new(machine, [[name(machine)], [supers([object])],
                     [dynamics([capability/1])], [statics([])]]).
```

When creating such a machine object, the capability of the machine is given by capability facts (the system answers by giving a unique identifier for the new object):

```
machine::new(M1,
         [[capability(oper1), capability(oper3), capability(oper5)]]).
M1 = o1
```

The assembly line object will also be an expert system which, given a sequence of operations, has rules to plan the sequence of machines that will be used to perform those operations. So the class `expert_assembly_line` uses multiple inheritance to combine behaviors of expert systems and composite objects. The rules select machines by their capability to perform the operation and their state (ok or down).

```
part_whole_class::new(expert_assembly_line, [
   [name(expert_assembly_line)],
                              % expert_assembly_line gets part-whole and
                              % and expert system behaviors from
                              % part_whole_object and expert_system_class
   [supers([part_whole_object, expert_system_class])],
                              % dynamic predicates will hold the parts
   [dynamics([machine1/1, machine2/1, machine3/1])],
   [statics([
                              % the assembly line answers requests about
                              % Machine's capability by sending to this
                              % part the message capability(Operation)
      [(capability(Machine, Operation) :-
         self::to_a_part(Machine:capability(Operation)))],
      [(machines(machine1)),     % machines/1 static predicates simply
       (machines(machine2)),     % returns the name of the three parts
       (machines(machine3))]     % to be able to send them messages
     ])],
   [parts([machine1:machine, machine2:machine, machine3:machine])]
  ]).

expert_assembly_line::new(O1, [
                               % rule/2 facts holds the rules of the
                               % expert system: rule(Head, Body)
   [(rule(plan([],[]), true)), % Here are the two rules to plan the
                               % sequence of machines corresponding to
                               % a sequence of operations
    (rule(plan([Op | Operations_list], [M | Rest_of_plan]),  % Head
          (machines(M),                                      % Body
           capability(M, Op),
           state(M, ok),
           plan(Operations_list, Rest_of_plan)))],
                               % goals capability/2 and machines/1 are
                               % directly callable in the embedding
                               %object
   [(callable(capability(_, _))),
    (callable(machines(_)))],
   [(askable(state(_, _)))],   % state/2, giving the state of a
                               % machine is askable to the user
                               % Next, names and initial clauses are
                               % given for the three machines
                               % composing the line
   machine1:[[capability(oper1),capability(oper3),capability(oper5)]],
   machine2:[[capability(oper1),capability(oper2),capability(oper3)]],
   machine3:[[capability(oper3),capability(oper4),capability(oper5)]]
  ]).

O1 = o4                        % O1 is unified to the object identifier
```

Querying the object is simply done by sending to o4 the message solve(plan(<list of operations>, P)) and P will give the solution: the list of machines to use in the processing. In the first query, when prompted by the expert system, the user answers that all machines are up but in the second one, he says that the machine 1 is down so the planner does not use this machine in its plan.

```
o4::solve(plan([oper1,oper3,oper4],P)).
state(machine1,ok)? yes.
state(machine3,ok)? yes.

plan([oper1,oper3,oper4],[machine1,machine1,machine3]) is
proved using the rule
IF machines(machine1) AND capability(machine1,oper1) AND
state(machine1,ok) AND
plan([oper3,oper4],[machine1,machine3])
THEN plan([oper1,oper3,oper4],[machine1,machine1,machine3])

machines(machine1) is true because it is callable and the
call succeeded

capability(machine1,oper1) is true because it is callable
and the call succeeded

state(machine1,ok) is true because you told me so
...

P = [machine1,machine1,machine3]


o4::solve(plan([oper1,oper3,oper4],P)).
state(machine1,ok)? no.
state(machine2,ok)? yes.
state(machine3,ok)? yes.

plan([oper1,oper3,oper4],[machine2,machine2,machine3]) is
proved using the rule
IF machines(machine2) AND capability(machine2,oper1) AND
state(machine2,ok) AND
plan([oper3,oper4],[machine2,machine3])
THEN plan([oper1,oper3,oper4],[machine2,machine2,machine3])
...

P = [machine2,machine2,machine3]
```

This simple example gives an idea of the power of ObjVProlog and of the flexibility arising from the use of metaclasses.

5. APPLICATIONS OF METACLASSES

The techniques used for defining part-whole hierarchy and expert system objects can serve to define a lot of different object behaviors. It suffices to provide a metaclass that allows the creation of particular classes to instantiate such objects and a class equivalent to object which implements the behavior of these objects when a message is received. Here are examples of use.

5.1 Speedy classes

In all object-oriented language, the dynamic binding of a message to the method that will be used to answer it is time consuming. Cache mechanisms can be used to speed this process.

One can implement these mechanisms through a metaclass `speedy_class` which instantiates classes that remember (using a hash table, for example) where clauses used to answer some messages are located in their superclasses hierarchy. Space, in the case of these classes, can be traded for speed.

5.2 Parallel object

Parallelism based on the parallel object model can also be implemented by metaclasses. As a first step, message passing may be defined in `object` and inherited by all objects in the system. Then, we could instantiate a class `parallel_object` that would define parallel message passing between parallel objects. In the context of distributed programming the use of classes may lead to inefficiencies due to repeated accesses from objects to their instantiating classes. A way to solve this problem could be the definition of a metaclass `parallel_class` that would provide classes with replication mechanisms over all distributed sites.

5.3 Inference engine

The expert system example showed how different objects may execute using different interpreters. This idea can be pursued further with a metaclass allowing the creation of classes defining their own interpreter. For example, a class could instantiate objects where goals are reduced in a breadth-first order instead of depth-first. Another one could instantiate objects working in first order logic or even working with a subset of Prolog, like having only facts in the object's database, to gain efficiency.

5.4 Actor-like declarative state

Actually, ObjVProlog implements objects' state as clauses and provides "assume" and "forget" to change them. We are considering the possibility of implementing state modification of an object by the creation of a new version of the object, with an underlying temporal semantics in mind. However, full copying of an object for each modification may waste to much space. Virtual copy mechanisms would solve the problem and they can also be implemented using a metaclass to create objects that create a copy for only the modified predicates and retrieve clauses from appropriate versions when receiving a message.

6. RELATED WORKS

An object of ESP [CHIK84] is a collection of axioms and a message is a request to refute the submitted proposition using these axioms. However, unlike ObjVProlog, the state of the object is procedural and implemented as slots with destructive assignment and no metaclass mechanism is provided. Spool [FUKU86] is essentially in the same vein, building classes with block-structured syntax and implementing state via imperative instance variables. PAL [AKAM86], Phocus [CHAN87] and CIEL [GAND88] introduce the notion of type in languages similar to Prolog. Classes serve as complex type constructor but again, metaclasses are not provided. In ObjVProlog, no notion of type is involved, classes serve as state and behavior declaration for entity with a coarser grain.

Probe [GAND87] is also a language based on the ObjVLisp model but makes choices

different from ObjVProlog. If classes have methods defined by Prolog-like clauses, objects' states are implemented by imperative attributes. The semantics of inheritance is monotonic and relies on the "cut_inheritance" to force non-monotonic behaviors when necessary. As said earlier, inheritance in ObjVProlog has a non-monotonic semantics and relies on "super" to access superclasses' shadowed definitions.

Some proposals provide parallelism. Object-Prolog defines worlds as collections of Prolog-like clauses and a message is a request to prove a goal using these clauses [DOMA86]. Many worlds can run in parallel and parallel message passing is provided. Unlike ObjVProlog, no particular inheritance discipline is built into the language. Communicating Prolog Units take a similar approach but add meta-units to define how units will respond to messages or, eventually, forward them to another unit in an actor-like way [MELL87].

Some applications discussed in this paper can be done in Object-Prolog or in Communicating Prolog Units especially using its meta-units feature, but they are not class-based languages. ObjVProlog takes the class and metaclass approach and shows that this approach can be more convenient.

7. CONCLUSION

This paper describes a new object-oriented logic programming language based on the ObjVLisp model and Prolog. In this model, classes are proper objects and metaclasses are provided. We have shown how ObjVProlog can tackle some well-known problems of object-oriented and logic programming efficiently and elegantly. Metaclasses turn out to be a very powerful abstraction mechanism: part-whole hierarchy and expert system objects were implemented using them. ObjVProlog is ideal for this exploration since it combines the metaclass mechanisms of ObjVLisp and the ease of meta-programming of Prolog.

ObjVProlog is still being worked on. Extension are sought to make it more powerful in a clean and reflexive way. Potential uses of metaclasses will be to improve ObjVProlog with classes having methods cache, with metaclass definition of parallelism and virtual copy mechanisms or with classes instantiating objects executing in different logics by providing appropriate interpreter.

AKNOWLEDGEMENTS

The authors wish to thank Patrice Boizumault and Nicholas Graube for fruitful discussions and NSERC Canada for its financial support.

REFERENCES

[AKAM86] Akama, K. **Inheritance Hierarchy Mechanism in Prolog**, *Proc. Logic Programming '86 LNCS* **264** (1986), pp. 12-21.

[BLAK87] Blake, E. and Cook, S. **On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk**, *Proc. of ECOOP'87, Bigre+Globule* **54** *(June 1987), pp. 45-54.*

[BOBR88] Bobrow, D.G. et. al. **Common Lisp Object System Specification X3J13 Document 88-002R**, *Sigplan Notices Special Issue* 23 (September 1988).

[CHAN87] Chan, D., Dufresne, P., and Enders, R. **PHOCUS: Productions Rules, Horn Clauses, Objects and Contexts in a Unification-based System**, *Actes du Sém. Prog. en Logique, Trégastel* (May 1987), pp. 77-108.

[CHIK84] Chikiyama, T. **Unique Features of ESP**, *Proc. Int'l Conf. on Fifth Gen. Comp. Sys.* (1984), pp. 292-298.

[COIN87] Cointe, P. **Metaclasses are First Class: The ObjVlisp Model**, *Proc. of OOPSLA'87, ACM Sigplan Notices* 22, 12 (December 1987), pp. 156-167.

[COIN88] Cointe, P. **A Tutorial Introduction to Metaclass Architecture as Provided by Class Oriented Languages**, *Int'l Conf. on Fifth Gen. Comp. Sys., Tokio 1988* (1988).

[DOMA86] Domán, A. **Object-PROLOG: Dynamic Object-Oriented Representation of Knowledge**, *SzKI Comp. Research and Inn. Center* (1986).

[FUKU86] Fukunaga, K. and Hirose, S. **An Experience with a Prolog-based Object-Oriented Language**, *Proc. of OOPSLA'86, ACM SIGPLAN Notices* 21, 11 (November 1986), pp. 224-231.

[GALL86] Gallaire, H. **Merging Objects and Logic Programming: Relational Semantics**, *Proc. of AAAI '86* (August 1986), pp. 754-758.

[GAND87] Gandilhon, T. **Proposition d'une Extension Objet Minimale pour Prolog**, *Actes du Sém. Prog. en Logique, Trégastel* (May 1987), pp. 483-506.

[GAND88] Gandriau,M. **CIEL: Classes et Instances En Logique**, *Thèse de Doctorat, ENSEEIHT* (1988).

[LEON88] Leonardi, L. and Mello, P. **Combining Logic and Object-Oriented Programming Language Paradigms**, *Proc. 21st Hawaii Int'l Conf. on Sys. Sc.* (1988), pp. 376-385.

[MALE89] Malenfant, J. **ObjVProlog: un modèle uniforme de métaclasses, classes et instances adapté à la programmation logique**, *Département d'inf. et de r.o., Université de Montréal, Rapport de recherche* 671 (January 1989), 58 p.

[MELL87] Mello, P. and Natali, A. **Objects as Communicating Prolog Units**, *Proc. of ECOOP'87, Bigre+Globule* 54 (June 1987), pp. 233-243.

[SHAP83] Shapiro, E. and Takeuchi, A. **Object-Oriented Programming in Concurrent Prolog**, *J. of New Generation Computing* 1, 1 (July 1983), pp. 25-48.

[STER86] Sterling, L. and Shapiro, E. **The Art of Prolog**, MIT Press (1986).

[TARA88] Tarau, P. and Boyer, M. **Prolog Meta-Programming with Soft Databases**, *Proceedings of the Work. on Meta-Programming in Logic Prog. (J.W. Lloyd ed.)* Bristol (1988), pp. 269-280.

[TOUR86] Touretsky, D. **The Mathematics of Inheritance Systems**, *Morgan Kaufmann* (1986).

[VAUC88] Vaucher, J., Lapalme, G. and Malenfant, J. **SCOOP: Structured Concurrent Object-Oriented Prolog**, *Proc. of ECOOP'88, Springer-Verlag LNCS* 322 (August 1988), pp. 191-211.