EXTENDING LOGIC PROGRAMMING TO OBJECT PROGRAMMING : THE SYSTEM LAP

Herman ILINE & Henry KANOUI

Institut International de Robotique et d'Intelligence Artificielle de Marseille

13241 Marseille Cedex 1 France

ABSTRACT

Object oriented programming aims at code lisibility and conciseness via abstract data type declarations and takes advantage from setting the objects (ie data) in the center of the application, instead of procedures or demonstrations.
Just as in Logic programming, where describing the properties of the solution of some problem is (theoreticaly) enough to allow the program to compute the solution, Object programming proceeds from the fact that purely declarative information leads to more procedural behaviour.

It appears that Logic programming and Object programming are complementary in the sense that the first is very suitable for expressing subtle reasoning, but rather weak as a formalism for describing complex structures, and conversely for the second.

These considerations led us, at the IIRIAM, to combine Logic and Object programming in order to use the advantages of both. Our research effort resulted in the system LAP, an extension of Prolog to object programming, giving a new dimension to the latter, through logic programming concepts: processing of partially known data and non determinism applied at the object level.

The first part of this paper introduces the main characteristics of LAP, faced with the requirements of AI application development. The second part illustrates some of these capabilities through simple examples.

I INTRODUCTION

Practical applications of Artificial Intelligence, and particularly Knowledge Based Systems (or Expert Systems) involve two key points: the representation of knowledge and reasoning modules; these will link together facts and expert rules to construct the solution of the problem in a dynamic process.

In fact, faced with a concrete problem, at least two preliminary questions arise:
- what is the nature of the domain to be examined, what are the entities which compose this domain, what type of relations link these entities? Answering this first series of questions will help us to decide which knowledge representation schema to adopt.
- what kind of reasoning is to be performed on these entities? This will help to choose amongst inference engines and deduction methods.

Moreover, considering these questions will allow us to determine where the expertise is located: do we have to perform very subtle reasoning on flat and independant facts or, on the contrary, are we in the presence of deeply interconnected entities, the expertise consisting in dealing with these interactions? In other words, is it the power of the reasoning modules or the ability to describe complex universes which determines the choice of the tools and techniques?

In the second case, it is likely that object oriented programming will be a valuable tool to help in solving the problem. We can distinguish three levels of ambition of an object oriented programming language designer:
- hierarchical modeling exploiting the inheritance mechanism.
- communication between 'intelligent' entities.
- the construction of an abstract universe including specialized knowledge bases in order to reason on concrete objects.

The first approach (almost purely technical), aims at code lisibility and conciseness via abstract data type declarations. The second one (more 'animated') takes advantage from setting the objects (ie data) in the center of the application, instead of procedures or demonstrations. The third approach (referred to as cognitive) considers abstract objects as nodes where pieces of knowledge are centralized.
One could argue that the intelligence transmitted by this last approach proceeds from the fact that purely declarative information leads to more procedural behaviour, just as in logic programming where describing the properties of the solution of some problem is (theoreticaly) enough to allow the program to compute the solution.

The execution of a program written in an object oriented language can be seen as a communication process between objects coming up from a hierarchical universe of abstract frames. During the life of an object , its dynamic image is kept consistent with the image of every frame with which the object is concerned.

Now, from a user's point of view, object oriented programming can be a valuable tool for:
(a) Structuring (modeling) a domain;
(b) Maintaining the consistency of the universe of frames;
(c) Making some superficial deductions on this universe, using the inheritance mechanism;
(d) Reasoning on this universe by activating more general inference mechanisms.

Depending on the respective importance of these aspects, one can distinguish applications such as:
(1) Structuring an abstract universe:, which invokes aspects (a,b) above; we are only interested in describing the universe.
(2) Management of a domain of concrete objects; we are interested in exploiting the description of the universe by invoking aspects (a,b,c).
(3) Expert systems which need, above all, reasoning capabilities taken from aspect (d).

Applications of type (1,2) are most commonly achieved by object oriented programming. For applications of type (3) this approach eliminates from the knowledge bases all 'superficial' rules based on inheritance or directly derived from descriptive information attached to the objects. There only remain truly expert rules for which inference engines are still to be developed. To achieve this, object oriented environments based on algorithmic (C, Pascal) or even symbolic (Lisp) languages need a great deal of programming effort and are not well adapted to this purpose. Concerning the inference engines, those based on the production rules schema are very limited : the lack of semantics and the approximative modeling of reasoning are paid for an inflation of the number of rules and yet obtaining an uncertain (sometimes doubtful) result.

On the contrary, since we aim to formalize reasoning, we must have recourse to the mathematical logic and theorem proving which led to the so-called 'logic programming' of which Prolog is the only effective implementation. This provides us with a powerful formalism based on the notion of relation, a very precise semantics, and deduction strategies allowing the manipulation, in a non deterministic way, of very general data structures (trees with variables).
However, if Prolog is very suitable for expressing subtle reasoning, it is rather weak as a formalism to describe complex structures. This led us, at the IIRIAM, to combine logic and object programming in order to use the advantages of both. Our research effort resulted in the system LAP.

LAP is an extension of Prolog to object programming, giving a new dimension to the latter, through logic programming concepts: processing of partially known data and non determinism applied at the abstract object level:
• Objects and their slots are processed as true Prolog variables: definition of constraints on the values of object slots, processing of partially known objects. The non determinism of Prolog offers an elegant and efficient means for such an approach which allows a high degree of versatility compared with imperative implementation languages.
• The query language of the objects data base is Horn-clause logic and is expressed in terms of relations to be satisfied (as in Prolog, ie without distinction between input and output arguments).
• The deduction capabilities provided at the object level (facets and methods) are increased by the power of Prolog as an inference engine by itself, as well as a programming language.

Thus, LAP reassembles concepts usually dispersed between:
- relational data bases (relations between frames),
- object oriented languages (parentship, inheritance, message passing),
- inference engines (slot value deductions, facets and methods),
- logic programming (non determinism, variables).

In the second part of this paper, we will expand upon the main characteristics of LAP and show how they fit the considerations given above. This presentation is illustrated by examples given in Quintus-Prolog syntax.

II THE MAIN CONCEPTS OF LAP

LAP provides two distinct environments: the first one is for conceiving a universe of abstract frames, which we call **models**. In the context of this universe, the second environment controls the construction, evolution and reasoning of a realization, which is a set of concrete objects (or **instances**) compatible with the universe's constraints. In the center of these two environments is the notion of a model and object image.

To achieve this, LAP provides a set of library predicates for creating, modifying and maintaining the consistency of a universe of objects in which:
- a concrete object is described by abstract models and inherits their generic properties.
- An abstract model is part of a hierarchy ordered by the parentship link.
- Consistency constraints, default reasoning, specific processing (methods) can be attached to a model and inherited by its instances.
- An expert system application can be seen as a sequence of goals, formulated through a dialogue, and their demonstration.

The image of a model results from:
- its position in the graph of models which supports inheritance.
- Its relationships with other models which is another means of structuring the abstract universe.
- The declarative attachments: slots, their constraints and properties.
- The procedural attachments: expert rules, facets, methods.

The image of an instance is restricted to the relations with other objects, and values assigned to slots or deduced by expert rules or inherited from the models prototyping the instance.

In LAP, the models, the slots and their properties, the facets, methods and relations are explicitly declared. Each declaration results in Prolog assertions. Facets and methods must be edited as Prolog clauses.

A. Structural Concepts: the Graph of Models and the Relations

A model is an abstract data structure corresponding to a class of objects in the modelized universe and defined by a set of common properties. A given model may have one or several father models (direct ancestors). Thus LAP allows multiple parentship at the model level.

A parentship link between two models can be understood in two different ways:
- Either this link indicates that the father-model is a super-class of the son-model. For example, the model 'motor-car vehicle' is a son of the model 'land vehicule'.
- Or the father-model corresponds to a particuliar point of view of the son-model. For example, a 'motor-car' can be considered under two different points of view as 'transportation means' and 'assembly of metal parts'.

There is no fundamental difference between the two interpretations 'super-class' and 'point of view': in the two cases the son-model inherits all the characteristics attached to the father-model. However, the second interpretation will be preferred in those cases where we do not intend to create concrete instances of the model considered as 'point of view'.

The universe is given a graph structure whose nodes are the abstract models (to which knowledge is attached) and the vertices, denoting the parentship links, are the only vehicles of inheritance.

In addition to parentship, two models can be linked in LAP by one or more binary relations (as classic relations in relational data bases).These relations are another way to structure the universe and give access to the linked objects.

Let us now consider the example of a universe including human beings, wild animals, and domestic ones. Here is a possible modelization of this universe with some 'natural' relations.

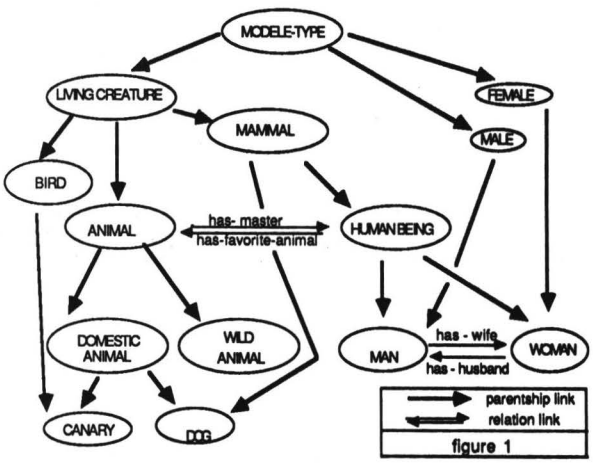

figure 1

The model and relation names are Prolog atoms. The models 'human' and 'animal' are linked by the relation 'has_favorite_animal', the symmetric relation being 'has_master'. Equally the models 'man' and 'woman' are linked by the relations 'has_wife' and 'has_husband'.
The model 'canary' has two father-models:
- 'bird' interpreted as a super-class: every canary is a bird.
- 'domestic_animal' will rather be interpreted as a point of view on 'canary'.

In this example we do not intend to manipulate instances of 'domestic_animal', but the associated point of view will be useful dealing with an instance of 'canary' or 'dog'.
A similar remark can be made for the points of view 'male' and 'female' for the models 'man' and 'woman'.

The data base can be constructed step by step by invoking LAP primitives. For example,

```
| ?- def_model(canary,[bird,domestic_animal]).
```

defines 'canary' as a descendant of the ordered set of models 'bird' and 'domestic_animal'. Note that the order between ancestors is used in solving conflicts arising from multiple inheritance.

The parentship links can now be interrogated by the primitive 'fat_son' which links a model to its direct descendants. For example:

```
| ?- fat_son(X,canary).
X=bird ;
X=domestic_animal ;
no
```

Now, if the model 'human_being' is assumed, in order to define the binary relation 'has_master' between 'human_being' and 'animal', we have to type:

```
| ?- def_link(animal,[has_master,[0,1]],
        [has_favorite_animal,[0,5]],human_being).
yes
```

which indicates that an animal has at most one master and introduces the symmetric relation 'has_favorite': a human being has at most five favorite animals.
Here, we just indicate the possibility that a relation can hold between humans and animals. The link will be effective at the instance level (in the concrete world) only.

The primitive 'link' gives the relations which can link two models. For example:

```
| ?- link(animal,X,human_being).
X=[animal,[has_master,[0,1]],human_being]
```

The term substituted for the variable X indicates the origin (through inheritance) of the link. If several links exist between 'animal' and 'human_being', they will all be printed.

To know which models are directly linked (without passing through inheritance) by the relation 'has_master', one could ask:

```
| ?- fat_son(domestic_animal,Y).
Y=canary ;
no

| ?- fat_son(X,Y).
X=animal, Y=domestic_animal ;
X=bird, Y=canary ;
X=domestic_animal, Y=canary ;
   ......
no
```

It must be noticed that when invoking LAP primitives, as is usual in Prolog, there is no distinction between input and output arguments. All the substitutions for the free variables which satisfy the relation are enumerated by backtracking. This feature is not usual in other object oriented languages; it is directly inherited from the underlying Prolog system.

One can also use the primitive 'anc_off' which is a generalization of 'fat_son' and links a model to all its descendants, direct or not:

```
| ?- anc_off(X,canary).
X=bird ;
X=living_creature ;
X=modele_type ;
X=domestic_animal ;
X=animal ;
no
```

The order in which the results are given corresponds to a depth-first search strategy as in Prolog. This order is important: recall that the multiple inheritance schema can lead to a conflict when some information is searched for among the ancestors of a model. This conflict is solved by enumerating the ancestors by a depth-first strategy and retaining the first which provides the data searched for.

```
| ?- link(X,[X,[has_master,A],Y],Y).
X=animal, A=[0,1], Y=human_being ;
no
```

The existence of a relation between two models is transmitted to their respective descendants. So, to ask: "Which are the relations which link the model X, descendant of A, to the model Y, descendant of B?", one could use:

```
| ?- link(X,[A,R,B],Y).
X=animal,A=animal,R=[has_master,[0,1]],
B=human_being,Y=human_being ;
X=domestic_animal,A=animal,R=[has_master,[0,1]],
B=human_being,Y=human_being ;
X=canary,A=animal,R=[has_master,[0,1]],
B=human_being,Y=human_being ;
X=dog,A=animal,R=[has_master,[0,1]],
B=human_being,Y=human_being ;
X=wild_animal,A=animal,R=[has_master,[0,1]],
B=human_being,Y=human_being ;
X=animal,A=animal,R=[has_master,[0,1]],
B=human_being,Y=man ;
.......
X=wild_animal,A=animal,R=[has_master,[0,1]],
B=human_being,Y=man ;
.......
X=wild_animal,A=animal,R=[has_master,[0,1]],
B=human_being,Y=woman ;
no
```

that is the complete set of combinations between 'animal' and its descendants on one hand, and 'human_being' and its descendants on the other hand, which is in complete accordance with the philosophy of Prolog.

B. Descriptive Concepts: Slots and their Values

The parentship links and the binary relations as shown in the graph of figure 1 reflect only the structure of the modelized domain. To each model, node of the graph, knowledge is attached by means of declarative and procedural characteristics which will express both the internal structure of the model itself, and its behaviour and relations with its environment. These characteristics will later be interpreted at the realization level by the instances of the model.

The most elementary way of expressing the descriptive knowledge associated with a model consists in providing it with a collection of **slots**. A slot defined for a model is also implicitly defined for all the descendants of the model, via the inheritance mechanism.
In LAP, a slot behaves like a Prolog variable and the values it can take are 'bound' terms (trees) in the Prolog sense. The set of possible values of a slot can be governed by a set of constraints and obey certain properties as decided by the programmer.

The constraints associated with a slot at the model level make it possible to specify:
- The **type** of the slot: integer, real, string, list, ... and more generally any reference to type testing available in the underlying Prolog system.
- The set of **legal values** : the one that the slot can take at the instance level.
- The slot **cardinality**. In LAP, slots are in general multi-valued. The cardinality indicates the maximum number of values that the slot can take simultaneously for a given instance. As usual in Prolog, the set of solutions will be enumerated by backtracking.
- The possibility of **inferring** slot values via a binary relation.

Apart from the constraints, one can define, at the model level, properties concerning the slot values. As for constraints, these properties will be effective at the instance level only. One can:
- Allow the **prompting** of a slot value, if it has not been possible to infer it by other means.
- Specify **default values** for a slot. These default values will be taken into account at the instance level only if all other attempts to compute the slot values failed to reach the slot cardinality.

The properties and constraints above were attached at the model level. Some additional properties can be attached directly at the instance level; they make it possible:
- To forbid certain values for a slot.
- To fix a slot value (forbid further modification).
- To declare unknown a slot value which forbids asking the user for it.

Given a model already defined, we can provide it with a new slot with the primitive 'add_slot'. For example, we can define slots 'name' and 'home' for model 'human_being' above by typing:

```
| ?- add_slot(name,human_being).
yes

| ?- add_slot(home,human_being).
yes
```

These slots can then be forced to obey certain constraints:

```
| ?- set_slot_property
        (type,[human_being,name,string]).
yes
| ?- set_slot_property
        (cardinal,[human_being,home,2]).
yes
| ?- set_slot_property
        (type,[human_being,home,string]).
yes
```

which indicates that the values of 'name' and 'home' are strings; the name is single-valued (default cardinality=1) but not the home (two possible addresses).

These slots can be prompted, which allows the system to question the user on their values. This is achieved by:

```
| ?- set_slot_property
 (prompt,[human_being,name],default_prompt).
yes
| ?- set_slot_property
   (prompt,[human_being,home],ask_address).
yes
```

which indicates that name and home address can be computed by deleting (in the Prolog sense) the literals 'default_prompt' and 'ask_address' respectively.
'default_prompt' and 'ask_address' refer to Prolog clauses. 'default_prompt' is provided by LAP as a library predicate, while 'ask_address' is a user-defined predicate.

Now, the properties of a model slots can be interrogated by:

```
| ?- get_slot_property(X,[human_being,A,V]).
X=cardinal, A=name, V=1 ;
X=type, A=name, V=string ;
X=prompt, A=name, V=default_prompt ;
X=cardinal, A=home, V=2 ;
X=type, A=home, V=string ;
X=prompt, A=home, V=ask_address ;
no
```

C. Procedural Concepts: Facets and Methods

Having specified the links and relations of a model and also described its descriptive features, we are now interested in expoiting this structure. In fact, apart from the usual inheritance mechanism, the use of slots and their values is governed by ad-hoc rules which constitute knowledge bases associated with the models. In LAP, there are three kinds of such rules:
- rules associated to events,
- rules for inferring slot values or relations,
- methods.

The first two kinds of rules are implemented as **facets**. The last is the software support for message-passing and is nothing other than a generalization of the first two.

The generic events addressed by the first kind of rules are:
- creation or suppression of a model instance: facets **if_created** and **if_suppressed**.
- assignment or suppression of a value to an instance slot : facets **if_added** and **if_removed**.

The second kind of rules, the deduction rules, correspond to the facet **if_needed**.

At each attempt to provoke one of the events above, the corresponding facet - provided it has been defined for the model on which the considered instance depends - is fired. The event under consideration will effectively occur only if the Prolog goal corresponding to the facet can be satisfied.

The role of the rules associated to the events is to preserve the integrity and consistency of the object data base when it is modified or updated.
The facet **if_needed** makes it possible to express how the values of an instance slot can be inferred from other information. The inferred values will then simulate the event **if_added**. So, a value deduced by the activation of rules, but not consistent with the aspect if_added will be rejected.

Facets and methods are written in Prolog and it is important to recall that the conceptor of an application disposes of the full Prolog power to express a facet or a method. For example, the facet **if_needed** defined for a given slot of an instance can not only invoke the value of other slots of the considered instance, but also refer to other objects (via LAP primitives), which makes it possible to chain rule activation.

III REALIZATION: THE CONCRETE UNIVERSE

The concrete universe is a collection of instances. The image of an instance is essentially constituted by the set of known slot values.

The life of an instance is nothing other than a succession of events including:
- the creation or the destruction of the instance,
- the setting of binary relations with other instances,
- the modification of slot values,
- the interrogation of slot values (by reading assigned values, rule inferences, deductions through relations, user inputs, or by returning default values).

Taking these events into account is controlled by knowledge rules (ie facets, methods,...) embedded in the respective model images.

LAP offers primitives to read or write, for the benefit of a given instance, the values of the slots known by the (possibly multiple) models on which the instance depends. As explained above, assigning and getting a value can be governed by programmer-defined procedures: the facets.

To illustrate these points, we will again consider the example given in figure 1. We will describe a scenario where an instance of the model domestic_animal is created.
(1) If, when creating a domestic animal, the name of its master is not specified, then the animal will send a message to every instance depending of an offspring of the model 'human_being'.
(2) The content of this message is : "do you want me as your favorite animal?"
(3) The addressees of the message will look at some slots (color, beauty,...) of the animal and will answer YES to the animal only if the corresponding values agree with their taste.
(4) To the human being which is ready to take care of it, the animal will send a second message whose content is: "where will you accomodate me?"
(5) The animal will accept as master the first human being who answers "inside the house" .

Assuming that methods corresponding to the messages have been defined for the model 'human_being', the implementation of this scenario requires:
- the facet if_created for the model 'domestic_animal',
- the facet if_needed for the relation 'has_master' in the model 'domestic_animal',
- and the Prolog clauses corresponding to these facets.

The facet declarations are achieved by:

```
| ?- def_facet([if_created,domestic_animal],
              create_domestic_animal).
yes
| ?- def_facet([if_needed,domestic_animal,
              has_master],find_master).
yes
```

The Prolog clauses 'create_domestic_animal' and 'find_maste'r could be written respectively as:

```
create_domestic_animal([domestic_animal,A],H,
        [A,K,get_slot(A,beauty, b)]) :-
    element([has_master, M],H), !,
    union(H,[beauty, b], K).

create_domestic_animal([domestic_animal,A],
        H,[A,K, get_and_set_slot(A,has_master,M)]).

find_master([domestic_animal,A],has_master,M):-
    anc_off(human_being, H),
    population(H,P),
    element(M,P),
    send([A, M], do_you_want_me,_,yes),
  send([A,M],where_would_you_accomodate_me,
              [inside,outside],inside), ! .
```

This simple example shows how:
- the effective creation of an instance can be forbidden if certain conditions (formulated in the Prolog rules defining the facet if-created) are not satisfied.
- As soon as an instance is created, some slots or relations can be assigned a value.
- The chaining of rules is achieved by invoking, from facets and methods, LAP primitives such as interrogating the objects data base and sending/acknowledging messages.

IV IMPLEMENTATION

LAP is implemented as a library of about 150 predicates described in detail in the LAP Reference Manual. It is available under Prolog II and Quintus-Prolog and runs under Vax/Vms, and Sun/Unix. Versions for Macintosh and Bull-SPS9 are under development.

LAP gives full access to the underlying Prolog system and provides a windowing and menu interface which facilitate both knowledge acquisition and execution.

LAP has been used in the development of several CAD systems in architecture, industrial simulation and medical consultation expert systems.

REFERENCES

1 Bobrow, D.G., and Stefik, M. "Object-Oriented Programming : Themes and Variations", AI Magazine 6, no. 4, 1986

2 Giannesini, F., Kanoui, H., Pasero, R., Van Caneghem, M. "PROLOG", Addison Wesley, 1986

3 Iline, H. "Un Environnement Orienté-Objets en Prolog : le Système LAP", Proceedings of Second CIIAM, Marseilles, France, 1986

4 Iline, H. "LAP: Manuel de référence", Internal paper, IIRIAM, 1986.

5 Shapiro, F., and Takeuchi, A. "Object-Oriented Programming in Concurrent PROLOG", New Generation Computing, 1983.

6 Stabler, P. "Object-Oriented Programming in PROLOG", AI Expert, 1986