OBJECTS AS COMMUNICATING PROLOG UNITS*

Paola Mello, Antonio Natali

DEIS - Facolta' di Ingegneria
University of Bologna-ITALY

ABSTRACT

The aim of this paper is to present a set of extensions to the Prolog language in order to insert in it concepts typical of parallel, distributed object-oriented systems. A program is a collection of objects (P-Units) that represent chunks of knowledge expressed as separate Prolog programs. P-units interact by asking for the demonstration of goals conceived as requests for operations. P-units can perform operations in parallel. Policies of interaction between objects, the creation of parallel activities, inheritance and delegation mechanisms are not frozen in the basic interpreter, but can be explicitly expressed in particular P-units that act as meta-objects. This approach enhances both the flexibility and reusability of the resulting object-oriented system.

1. INTRODUCTION

The object-oriented programming paradigm has shown itself to be very useful in a number of applications because of its protection and flexibility features. Systems based on the object model are intrinsically modular, easy to use and potentially open towards new multi-processor and distributed architectures.
On the other hand, logic programming languages [1], in spite of their very attractive declarative style and their expressive power, seem to be lacking in precisely the object-oriented characteristics mentioned above. In particular, in the most widely used language of this type, Prolog [2], it is difficult to express concepts such as modularity, protection and information hiding. Besides that, Prolog is a sequential language which does not allow programmers to exploit the advantages of multi-processor and distributed architectures, and, again, it is also very hard in logic to deal with problems which need the concept of state, such as the control of physical devices. Problems of this type have been solved in Prolog by introducing built-in, non-logical predicates.
The lack of object-oriented features could prevent Prolog from being used to develop large, complex software systems, even if its declarative style and efficient implementation [3] make it very attractive for several applications.
The main aim of the proposal presented in this paper (called CPU - Communicating Prolog Units - [13]) is to explore the usefulness and the synergetic advantages of

---

*This work has been supported by CEE Esprit Project p973, ENIDATA S.p.A. and Italian National Research Council (CNR)

the combination of logic- and object- oriented programming to overcome the limitations mentioned above.

The integration of logic and object-oriented programming has already been the subject of research. Perhaps the most interesting and complete proposals are based, at the current state of the art, on Concurrent Prolog [6], [7], a language that is intrinsically parallel and not compatible with Prolog.
Moreover, the concept of 'multiple worlds', providing a way to group together assertions about a concept and the possibility to describe inheritance of properties between concepts, has already been adopted in several proposals of extension to Prolog such as Prolog/KR [8] , ESP [9], MULTILOG [26], SPOOL [10], and [11]. However, in these proposals there is no possibility to express explicit parallel objects.

Instead, the aim of CPU is to fully subsume Prolog and allow parallel activities to be explicitly expressed. Besides, it exploits meta-programming [19] in order to enhance software flexibility and re-usability and make it easier to experiment with different object-oriented organizations, according to rapid prototyping techniques.

In summary, CPU introduces the following concepts:
a) the compact Prolog database (facts and rules) is split into a collection of separate Prolog programs called P_UNITs, assimilable to CLASSES. P-units are the basic construct for modularity, information hiding and knowledge structuring, but no pre-defined policy is introduced for these topics;
b) the behavior of a single P-unit is modelled as a set of parallel demonstration activities, each owning a local state (INSTANCES);
c) interactions between different P-units are expressed as requests of goal demonstration (MESSAGES);
d) explicit communication policies between different P-units and instances are expressed in terms of "meta-rules" written in "meta-P-units" associated to "object-P-units". Static and dynamic INHERITANCE mechanisms, frozen at the implementation layer in traditional object-oriented systems (e.g Smalltalk [4] and Flavors [5]), can be explicitly stated in meta-P-units.

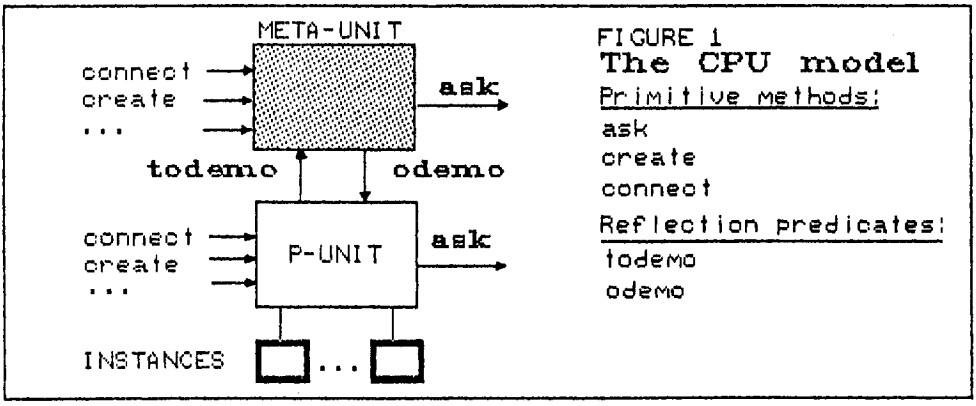
The impact of these concepts on the design and implementation of large software applications is currently investigated with reference to a programming environment for Prolog in the context of the ALPES Esprit project [12].
The paper is organized as follows. Section 2 introduces CPU classes, instances and basic communication mechanisms. Section 3 discusses other forms of communication between object instances. In section 4 the CPU approach to knowledge sharing between different objects is discussed.

2. THE CPU MODEL

2.1 CLASSES

In the CPU model [13] (see figure 1) a program is conceived as a collection of separate (and - if needed - distributed) objects, each one representing a chunk of knowledge on a particular domain. Each object (called P-unit) is expressed as a Prolog database [2]. A Prolog database is a collection of Horn clauses, i.e. logical implications of the form $A_0 :- A_1, A_2, \ldots A_n$ where A_i (i=1,..,n) is an atomic formula and n >=0. Variables occurring in atomic formulas are denoted by constants beginning with an upper-case letter, and are universally quantified.
Prolog clauses -that can be read declaratively as logical assertions- can also be interpreted procedurally: e.g. "to solve a goal (or to execute an operation suc-

cessfully) matching A_0, solve (execute successfully) the subgoals (operations) $A_1 \ldots, A_n$ in sequence". Backtracking can be activated to find the right solution if



more than one definition for the same operation exists.
Thus, in CPU, operations (methods) are simply represented by Prolog clauses. In the following example the P-unit 'list' containing methods to manipulate lists is presented. It defines the methods: 'append', 'insert' and 'notempty'.

```
unit(list).
append(...):-      ...
notempty(LIST):-   ...
insert([],ELEM,[ELEM]).
insert([FIRST|REST],ELEM,[FIRST|R1]):- isless(FIRST,ELEM),insert(REST,ELEM,R1 ),!.
insert([FIRST|REST],ELEM,[ELEM,FIRST|REST]).
isless(FIRST,ELEM):-...
```

P-units can communicate by sending messages; a message sent to a particular object can be conceived as a request to demonstrate a goal using a particular database. The primitive predicate ask(Dest, Goal, Res) embeds a synchronous message passing. It asks a P-unit 'Dest' to demonstrate 'Goal' with result 'Res'.
'ask' acts as a (remote) procedure call [17] where the caller waits until the invoked operation is terminated. 'Res' is bound to 'true' if the demonstration of 'Goal' into 'Dest' ends with success. In this case some of the variables within 'Goal' may have been bound to terms. In the case of failure, 'Res' is bound to any constant different from 'true'. The default value for a failure is 'unknown' and not 'false' because P-units are open-worlds [14]: the close world assumption of Prolog (which usually justifies a concept of falsity) must be established, if needed, in an explicit way.
If the method (goal) invoked is local to the current P-unit, the 'ask' primitive can be omitted. In the following example 'list' invokes the P-unit 'integer' to demonstrate the external goal 'isless'.

```
unit(list).
append(...):- ...
insert([],ELEM,[ELEM]).
insert([FIRST|REST],ELEM,[FIRST|R1]):- ask(integers,isless(FIRST,ELEM),true),
                                       insert( REST,ELEM,R1 ),!.
insert([FIRST|REST],ELEM,[ELEM,FIRST|REST]).
```

Expressing methods as Horn clauses and method invocations as goals to demonstrate, leads to a set of important features:
1) knowledge is expressed in a declarative style. This helps in building knowledge-based systems;
2) the use of goal arguments as either input- or output-parameters of operations is determined by the context at the execution-time thanks to unification [15];
3) the same method can have multiple definitions (see 'insert' in 'list'). In this case, the right solution is automatically searched via backtracking;
4) the system is not necessarily uniform: unification can be profitably applied to logical terms which are not represented as object instances.

2.2 INSTANCES

P-units can activate different and, if needed, parallel demonstration processes to solve goals (i.e. to perform requested operations). Thus, P-units can be considered as TYPE or CLASS objects whose instances are agents performing operations on behalf of an external sender object (MASTER) and sharing the knowledge base represented by the clauses of the class. The Basic Machine supporting a P-unit is conceptually a multi-processor system with shared memory and a finite but not limited number of processors.

In the P-unit 'list' presented before, a demostration process is implicitly activated every time a 'list' method invocation is received and the caller is suspended until such a process has produced an answer. List demonstration processes, like unserialized actors [16], have no private state. Thus, it is not necessary for the programmer to name them explicitly. Demonstration processes are only passive servers of external requests and can be completely transparent to the user.

However, although logic programming does not permit multiple assignments of values to logic variables, the instances of an object-oriented system normally have a private state that can be changed by the execution of methods. For this reason, CPU associates to each P-unit the primitive method 'create' in order to explicitly create parallel, independent INSTANCES with a private state.
When a P-unit 'u1' receives a request for the method: create(Iname, Goal), it creates an instance with name 'Iname' to perform the demonstration of 'Goal' using the Prolog data-base of 'u1'.
The 'create' method terminates with success just after the activation of 'Iname'. Thus, the instance 'Iname' performs the operation 'Goal' in parallel with its creator (the caller of 'create'). The names of explicit instances are represented as Prolog lists as follows: [unit_name, <unique-identifier>]. The state of an explicit instance is represented by the refutation-tree related to the goal to be demonstrated and by the local bindings of logical variables. As in object-oriented systems based on Concurrent Prolog, instance variables are represented by logical variables. CPU instances do not change their state by using assignment but by recursively calling themselves with different argument values, according to a side-effect-free style of programming. Prolog predicates such as 'assert' and 'retract' should not be used to obtain state change because they usually do not produce a modification on a specific instance, but a change in the whole Prolog database (Class) shared among all instances.

Let us consider the following example. A Class named 'Point' is defined whose instances represent points in a two-dimensional coordinate system. Each instance of this class has an instance variable, X, that represents its horizontal coordinate,

and an instance variable, Y, that represents its vertical coordinate. Each instance can assign different values to X and Y ('set-new'), show its current position ('show') or delete the point ('delete').

```
unit(point).
erase(X,Y):-    <on screen operations>.
move(X,Y,Z,W):- <on screen operations>.
display(X,Y):-  <on screen operations>.
/* this part of Point represents the methods 'erase', 'move' and 'display'*/

point(X,Y) :- serve(set-new(Z,W)),!,move(X,Y,Z,W),point(Z,W).
point(X,Y) :- serve(show(X,Y)),display(X,Y),!,point(X,Y).
point(X,Y) :- serve(delete),!,erase(X,Y).
/* this part of point represents the definition of its main loop; the arguments
of point predicate (X and Y) represent the instance variables. The 'serve'
predicate will be explained in more detail in the following */

new(X,Y):-   repeat,point(X,Y).
/* the 'new' method is called to create a new point with initial state repre-
sented by X and Y actual values */
```

The main loop 'point(_,_)' is a specification of the behavior of a point: "A point with coordinates X and Y can serve a 'set-new' message and become a point on the screen with new coordinates Z and W; it can serve a 'show' message and maintain its old state; it can receive a 'delete' message and cancel itself".
When an object O1 wants to create a new point 'p1' with X=3 and Y=5 the following message has to be sent to the class 'Point':

ask(point,create(p1,new(3,5)),Res).

Once the new point has been created, the object O1 continues its computation without waiting for demonstration of the goal new(3,5).
An instance terminates when the refutation process fails or when it ends with success. In the example the latter situation occurs when a 'delete' message is served.

2.3. META-PROGRAMMING TO SPECIFY OBJECT BEHAVIOR

A main concept in CPU is that the specification of object interactions can be considered a typical domain of meta-programming [19]. To support this concept, the model introduces a clear distinction between objects and meta-objects and a mechanism to define precise relationships between them.
A meta-unit, 'mu' can be associated with an object-unit 'u' by executing the following primitive method with success in 'u': connect(mu).
Since 'mu'is a P-unit, it can be connected to another 'meta-meta-P-unit' and so on: for this reason meta-levels can grow indefinitely. When a unit is associated to a meta-unit, each instance of the object-unit performs a DEFAULT COMMUNICATION with its meta_unit for each (sub)goal to be demonstrated. This consists in asking the meta-unit for demonstration of the following goal :

todemo(Master, Instance, Current-U, Goal, Result)

where 'Instance' is bound to the name of the object-level instance that has to demonstrate 'Goal'; 'Master' is bound to the name of the unit for which the demonstration is executed (in general the name of the calling P-unit), and 'Current-U' is bound to the name of the object-level unit involved in the goal demonstration. 'Result' represents the result of the 'Goal' demonstration.

For example, let us assume that the explicit instance [point,p1] executes: ask(integers,isless(3,5),true). The following invocation is then automatically generated in the meta-unit -if it exists- associated with 'integers':

todemo(point, [point,p1] , integers, isless(3,5), true).

Successful execution of the method 'isless' is strictly related to the successful execution of 'todemo' with the specified arguments.

Transition from the object-level to the meta-level is only half of the communication pattern which allows the sharing of results between these two levels according to the reflection principle [18]. Transition from the meta-level to the object level occurs when a meta-unit sends to its object-unit the goal

odemo(Master,Instance,Unit,Goal,R)

which forces the demonstration process 'Instance' to solve 'Goal' (with result 'R') into the object-unit 'Unit' with the specified 'Master'.

In the following example the P-unit 'list' has been associated with the meta-unit 'meta-list'. 'List' has been modified in order to enclose interconnection policies in its meta-unit only, to enhance system modularity and flexibility.

A list of characters can now be implemented without changing the 'list' class code: it is only necessary to connect it with another meta-unit. "Generic" [20] classes can be easily defined in this way.

```
unit( meta-list ).
todemo(Caller,I,U,isless(A,B),Res):- ask( integers, isless( A,B ),Res ).
todemo( Caller,I,U,G, R ):-          odemo(Caller,I,U,G,R ).
```

todemo ↑ odemo ↓

```
unit(list).
insert([],ELEM,[ELEM]).
insert([FIRST|REST],ELEM,[FIRST|R1]):- isless(FIRST,ELEM),insert(REST,ELEM,R1),!.
insert([FIRST|REST],ELEM,[ELEM,FIRST|REST]).
```

Let us note that meta-P-units are different from meta-classes of object-oriented languages such as Smalltalk [4]. In fact, while the main task of meta-classes in Smalltalk is the general description of Classes, the main task of meta-P-units in this example is that of specifying relationships between objects. More generally, specification of a P-unit can be confirmed, modified, or completed by the associated meta-P-unit. According to this interpretation, the behavior of the instances of a Class 'c' is determined not only by the object P-unit 'c', but also by its associated meta-P-unit.

There are several reasons for investigating an object-oriented model in which each object communicates with its meta-level before doing any operation. In particular, even if Classes are represented as collections of methods, the use of meta-level allows the definition of a single method [21] to answer all messages that an object receives. This feature could be useful to trace or record all messages sent to an object in a history-list, to debug its behaviour or to redirect messages.

Moreover, concepts and mechanisms usually frozen in language notations (e.g. scope rules, exception mechanisms, synchronization and communication protocols) can be expressed by meta-rules.

Further examples of meta-level applications will be discussed in the following. In particular section 4 will be devoted to discussing the impact of meta-programming in the implementation of inheritance mechanisms.

3. DEFINING COMMUNICATIONS BETWEEN OBJECT INSTANCES

The 'ask' primitive is used to send goals to P-Units that behave like Classes. P-Units (and their associated meta-P-units) activate processes in order to perform goal demonstrations. However, as happens in traditional object models, the need to send messages to specific instances with an explicit name and a local state, is present. For example, to move the particular point [point,p1], an explicit message must be sent to it and not to its generic class.
Since CPU is intended for distributed programming, the CPU model does not allow the sharing of logical variables between different instances (even if they belong to the same class). For this reason, the interaction between instances is not expressed in terms of streams and annotated variables, as in object models based on Concurrent Prolog [6]. There is no need for explicit 'merge' operators, so an incremental style of programming is encouraged. In particular, low-level communication between instances can be performed by calling the methods of predefined 'queue' P_units. Producer instances can insert messages in a queue using an 'out' method while consumer instances can remove them by calling an 'in' method. Queues are objects with an internal state represented by the messages received and not yet removed. Because queues are here conceived as the building blocks of instance communication, they cannot be represented as explicit instances. Therefore, their state cannot be represented by logical variables. In CPU, queues of messages are simply represented by ordered collections of Prolog facts; 'in' and 'out' operations are implemented through side-effects using Prolog pre-defined predicates 'assert' and 'retract'. Inspection and manipulation of messages is performed in a very simple and expressive way, thanks to unification.
In order to make queues able to share methods but not their local state, queue methods can be defined in a 'meta-queue' unit. In this case object-units can be assimilated to instances and the shared meta-unit to their class.
Thus, each P-unit can behave as a queue if it is connected to a meta-unit including the following meta-rules:

```
todemo(Caller,Inst,U,out(Message),Res):-odemo(Caller,Inst,U,assert(Message),true).
/* a message is inserted in the object queue */
todemo(Caller,Inst,U,in(Message),Res):- odemo(Caller,Inst,U,retract(Message),true).
/* if it exists, a message is removed from the object queue */
```

Concurrent accesses to the database of a P-unit using assert and retract predicates are assumed to be disciplined through built-in synchronization protocols so that modification to the database is performed in a consistent way.

3.1 AN EXAMPLE: ASYNCHRONOUS COMMUNICATION

Let us consider the point example of section 2. An asynchronous communication between 'screen' instances, to display, change or delete points, and 'point' instances, can be established by writing the following meta-P-units:

```
unit(request-server).
todemo(Caller,Instance,U,serve(Goal),Res):-
                        !,ask(queue1,in(message(Source,Instance,Goal)),Res).
/* to serve an external request for a particular 'Instance', a request message
for such an 'Instance' must be removed from the queue. If no message exists, the
'Instance' backtracks. */
todemo(C,I,U,G,R):-odemo(C,I,U,G,R). /* other goals are solved locally */
```

```
unit(request-sender).
todemo(Caller,Instance,U,send(Dest,Goal),Res):-
          !,ask(queue1,out(message(Instance,Dest,Goal)),Res)
 /* to send a request to a particular 'Instance' a message for such an Instance is
 inserted in 'queue1' */
todemo(Caller,Instance,U,Goal,Res):-odemo(Caller,Goal,Res).
```

If we connect 'point' to 'request-server' meta-unit, 'screen' to 'request-sender' meta-unit, and 'queue1' to 'meta-queue' (figure 2), then the instances of 'screen' and 'point' classes communicate in an asynchronous way.

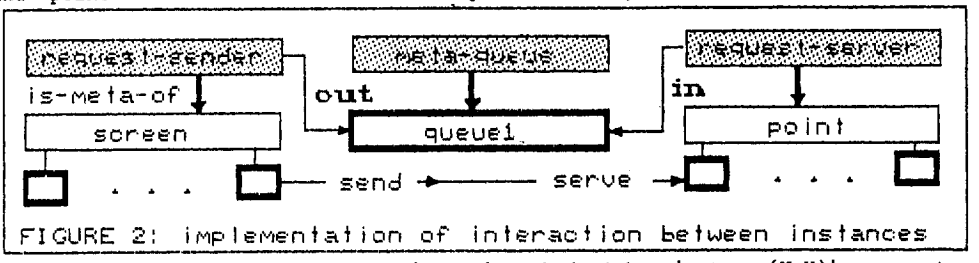


FIGURE 2: implementation of interaction between instances

When a point instance calls the 'serve' method with a 'set-new(X,Y)' argument, a request for the 'set-new' method is searched in 'queue1'. If this message exists it is extracted from the queue and executed; otherwise backtracking is induced in the 'point' instance and messages 'show' and 'delete' are sequentially searched for. If none of these messages is present in the queue the main loop is repeated and the 'serve' operations retried. Thus, 'point' instances perform a busy form of waiting for requests, using the backtracking mechanism of Prolog. Of course, if instances share the same processor this mechanism can be replaced by rescheduling policies.
Let us note the expressive power achieved through the unification mechanism for message-handling operations, when instance names are normal Prolog structures:
a) if 'Source' within the 'in' goal is not bound, a message can be received from any instance;
b) if 'Source' is partially specified (e.g. it has the form: [screen,X]), only messages sent by 'screen' instances can be received.
c) if 'Dest' is only partially specified within the 'out' goal, (e.g. it has the form [point,Any]) then the message is served by the first 'point' instance that executes a 'serve' predicate;
d) depending on the form of the requested operation ('Goal'):
   1) the first message in the queue is extracted;
   2) only a particular message is looked for.

The previous policies are examples of low-level process interaction mechanisms upon which different, higher level interaction policies (e.g. synchronous message passing) can be defined at meta-level. In [13] we have introduced a high form of interaction between instances called "synchronization clauses", based on the rendez-vous [20] model, similar to generalized clauses presented in [22]. Their implementation in terms of meta-rules is reported in [27].

3.2 CONTINUATIONS

When an instance terminates, it could send a last message with its final results to a 'continuation' [23] process, specified at its creation. For example, when each point instance terminates, it could send a special message to a particular 'garbage collector' process. This policy can be easily implemented without any modification

in the 'point' code, by simply writing the following meta-rule:

```
todemo(Master,Instance,U,new(Continuation,X,Y),R):-
    odemo(Master,Instance,U,new(X,Y),R),
    ask(queuel,out(Instance,Continuation,end(Instance,R)),true).
```

4. BUILDING INHERITANCE MECHANISMS

Very different mechanisms have been introduced in object-oriented languages for the purpose of sharing knowledge. In Smalltalk [4] simple inheritance is introduced; Flavors [5] supports multiple inheritance and combination; in Actors systems, delegation [21] is used. Since universal agreement seems impossible [24], the approach of CPU is to allow programmers to explicitly express rules to share knowledge between different objects. Experiments on different models of knowledge structuring and sharing can be performed in a rapid and flexible way since inheritance rules can be well expressed by meta-rules. Let us consider the following example. A 'special-point' class could be defined with the same behavior as 'point' (e.g. the same state representation, main loop, and sheduling policy) except a different 'move(X,Y,W,Z)' method. 'Special' point could be defined as a subclass of 'point' that only defines a new method for 'move'. Implementation of the required inheritance between these two classes can be expressed by an 'inheritance' shared meta-unit (figure 3).

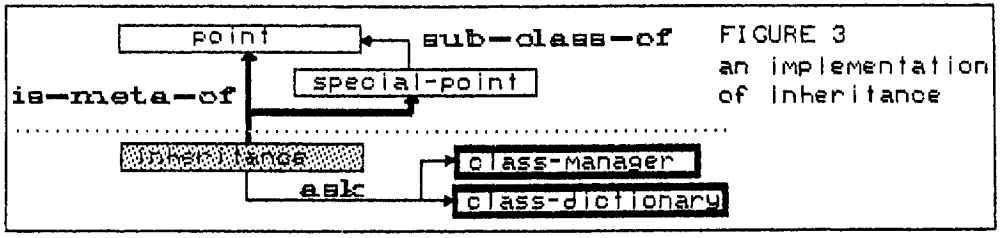


FIGURE 3
an implementation
of inheritance

If we suppose that the 'class-dictionary' unit specifies the methods defined in every class and the 'class-manager' states that 'super(special-point,point)', then meta-rules implementing inheritance can be written (in the order of their leading numbers) as follows.

A request for execution of a method 'G' by an 'Instance' of a 'Class' is accepted if 'G' is defined in it (rule 4); otherwise the execution of 'G' is delegated to the superclass 'SuperC' of 'Class' (if it exists) (rule 5).

```
4) todemo( C,Instance,Class,G,R):- ask(class-dictionary,def-method(Class,G),true),
                                   !,odemo( C,Instance,Class,G,R ).
5) todemo(C,Instance,Class,G,R):- ask(class_manager,super(Class,SuperC),true),
                                   todemo(C,Instance,SuperC,delegate(Class,G),R).
```

A request for execution in 'SuperC' of an inherited method 'G' is directly executed if 'G' is defined in 'SuperC' (rule 2); otherwise the execution of 'G' is delegated (with the same, initial subclass 'SubC') to the superclass 'SU' of 'SuperC', if it exists (rule 3).

Let us note that when a method is executed for a subclass, the Master is named in a special way (i.e. subclass(SubC)) to allow the selection of rule 1.

```
2) todemo(C,I,SuperC,delegate(SubC,G),R):-
                                   ask(class-dictionary,def-method(SuperC,G),true),
                                   !,odemo(subclass(C,SubC),I,SuperC,G,R ).
```

```
3) todemo(C,I,SuperC,delegate(SubC,G), R ):-
                                    !,ask(class_manager,super(SuperC,SU),true),
                                    todemo(C,I,SU,delegate(SubC,G),R ).
```

When a method invoked in a subclass 'SubC' is executed in a superclass 'SuperC', a local methods have to be solved by searching for the corresponding code in the inheritance tree, starting from 'SubC' (rule1). For example if, during execution of the 'point' main loop, the 'move' method is invoked, the corresponding code in the 'special-point' subclass must be executed. Let us note that a similar rule referring to instances is embedded at the implementation layer in traditional object-oriented systems like Smalltalk, and obtained by using the 'Client' variable [24] in delegation-based systems.

```
1) todemo(subclass(C,SubC),I,SuperC,G,R):- !,todemo(C,I,SubC,G,R).
```

If all the conditions previously mentioned are not verified, the result of the method invocation is the constant 'not-def' that can be interpreted as exception by the caller (rule 6).

```
6) todemo(C,I,U,G,not-def).
```

If a class has more than one super-class (e.g. when a new relation such as super(special-point,graphic) is written in the 'class-manager' unit) a multiple inheritance mechanism is automatically implemented: the inheritance tree is searched depth-first using the backtracking mechanism of the standard Prolog interpreter.
Using the same methodology more sophisticated inheritance mechanisms could easily be obtained. For instance, different dynamically-determined policies of knowledge sharing could be obtained depending on the method arguments or on the involved instances. We are still investigating the possibility of building delegation policies [21] using the basic CPU mechanisms introduced in the paper.

FINAL REMARKS

In this proposal, concepts typical of object-oriented models have been introduced in a logic programming framework based on the Prolog language. The use of meta-programming, typical of logic-based systems, allows programmers to build concepts and mechanisms usually frozen in specific language notations, e.g. communication and inheritance policies, in terms of meta-rules. The CPU model even though somewhat lacking in linguistic support, seems to be promising for defining highly dynamically reconfigurable systems and, in particular, flexible programming environments for rapid system prototyping. One of the objectives we will pursue in the Esprit ALPES project is to build a CPU working prototype and an environment kernel in which tools have a satisfactory degree of re-usability and flexibility in order to support rapid development of new abstractions in process interaction, deeply integrated with those related to knowledge representation. The first CPU implementation on a SUN machine showed the usefulness of combining object-oriented and logic programming models even if it still lacks in good performances, for the overhead due to the automatic transition to meta-level.
In order to enhance performances, we are going to investigate, among other things, the application of partial evaluation techniques [25] typical of logic programming languages.

REFERENCES:

[1] R.Kowalski: "Predicate Logic as Programming Language", Proc. IFIP-74

Congress. North Holland, pp. 569-574, 1974.

[2] W.F. Clocksin, C.S. Mellish : " Programming in Prolog ", Springer-Verlag, New-York, 1981.

[3] D.Warren et alii: "Prolog - The Language and its implementation compared with Lisp", SIGART Newsletter 64, pp. 109-115, 1977.

[4] A.Goldberg, D. Robson: Smalltalk-80, The Language and its Implementation. Addison Wesley, 1983.

[5] D.Moon et alii: "LISP Machine Manual", MIT-Books, AI Laboratory, 1983.

[6] K.Kahn: "Objects in Concurrent Logic Programming Languages", Proc. OOPSALA-86, Portland, Oregon, September 1986.

[7] K.Furukawa et alii: "Mandala: A Logic Based Knowledge Programming System", in International Conference on Fifth Generation Computer Systems, 1984.

[8] K.Nakashima "Knowledge Representation in Prolog/KR", International Symposium on Logic Programming, Atlantic City, February 1984.

[9] T.Chikayama: "ESP Reference Manual", ICOT Report, Feb. 1984.

[10] K.Fukunaga: "An experience with a Prolog-based Object-Oriented Language", Proc. OOPSALA-86, Portland, Oregon, September 1986.

[11] C.Zaniolo: "Object Oriented Programming in Prolog", International Symposium on Logic Programming, Atlantic City, February 1984.

[12] ALPES Esprit Project P973. Technical Annex 1985.

[13] P.Mello, A.Natali: "Programs as Collections of Communicating Prolog Units" In: ESOP, Saarbrucken, March 1986, Lecture Notes in Computer Science n.213, Springer-Verlag.

[14] C.Hewitt, P.De Jong :"Open Systems", Tech. Rep. MIT-AIM 691 December 1981.

[15] J.A.Robinson: "A machine-oriented logic-based on the Resolution Principle", Journal of ACM, 12, 23-41, 1965.

[16] D.G. Theriault: "Issues on the Design and Implementation of ACT2", Tech. Rep. MIT AI-TR 728, June 1983.

[17] B.J.Nelson: Remote Procedure Call. Dept. of Computer Science, Carnagie-Mellon Univ. , Ph.D. Thesis, Rep. CMU-CS-81-119, May 1981.

[18] K.Bowen, R.Kowalski :"Amalgamating language and metalanguage in logic programming", in Logic Programming , Academic Press, 1982.

[19] L.Aiello, G.Levi:" The uses of meta-knowledge in AI Systems", ECAI-84, Pisa, September 1984.

[20] "Reference manual for the Ada programming language ", U.S.Departement of Defense, ANSI/MIL-std 1815-a, Jan.1983.

[21] H.Lieberman: "Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-Oriented Systems", in J.Bezivin, P.Cointe (editors), 3eme Journees d'Etide language Oriente Objets, AFCET, Paris, 1986.

[22] M.Falaschi, G.Levi, C.Palamidessi: "A Synchronization Logic: Axiomatics and Formal Semantics of Generalized Horn Clauses." Information and Control, Academic Press, Jan.1984.

[23] R.E. Filman, D.P. Friedman: "Actors", in 'Coordinated Computing ', Prentice-Hall, 1984.

[24] H.Lieberman: "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems", Proc. OOPSALA-86, Portland, Oregon, September 1986.

[25] R.Venken: "A Prolog Meta-Interpreter for Partial Evaluation and its Application to Source to Source Transformation and Query-Optimization", in Proc. of ECAI-84, North Holland, 1984.

[26] H. Kauffman, A. Grumback: "MULTILOG: MULTIple worlds in LOGic programming", ECAI-86.

[27] P.Mello, A.Natali: "Configuration of Software Systems: A Domain for Meta-Programming", Technical Report DEIS, September 1986.