PROGRAMS AS COLLECTIONS OF COMMUNICATING PROLOG UNITS

Paola Mello (*)
Antonio Natali (*)

ABSTRACT

At the current state of the art, people are not encouraged to use logic programming languages for bulding large and complex software systems due to the difficulty of expressing concepts such as modularity, information hiding, data and process abstraction etc. In particular, it is impossible to build a system as a collection of different modules, independently designed and coded, with a lifetime extended till execution time.
This work is based on the fact that Prolog, the most used logical language, already has sufficient intrinsic power to express concepts of that kind.
Initially, a new mechanism is added to Prolog: the possibility to split a base of clauses into a set of different units. Subsequently, the distinction between object-level and meta-level units is introduced in order to express interactions between units. Finally, the possibility of specifying parallel flows of control within the same computation, is presented.
The main thesis of the work is that complex software systems can easily be built by fitting together a set of invariant Prolog units through a meta-level specification. Predicates can be viewed as communication channels statically or dynamically connected by meta programs. Such a methodology, which is suggested in quite a natural way by the declarative style of Prolog, could be fundamental also when units are written 'in-the-small' using other styles of languages.

(*) D.E.I.S., Faculty of Engineering, University of Bologna,
V. Risorgimento 2, 40136 Bologna, ITALY

This work has been supported by ENIDATA S.p.A and by Italian M.P.I.

INTRODUCTION

The use of very high level languages as tools for software development is nowadays a fundamental practice for enhancing software productivity and maintenance. Programming languages are usually classified according to three fundamental, different 'styles'; imperative, functional and logical.
Although lively competition exists among the supporters of each style, motivated by the great impact that different concepts and mechanisms have on software design, the different schools substantially agree upon a set of concepts intrinsically related to the organization of large and complex software systems.
Modularity, information hiding, data and process abstraction are fundamental issues in each programming style for building systems made of collections of components able to interact in a disciplined and protected way.

Historically, the first step in this direction was the design of programming languages and environments able to recognize and handle modularity at translation time only. At run time, most of the separation and protection among components was lost. Subsequently, thanks also to the fact that hardware costs had steadily gone down, the life of modules and protection was extended at execution time too. Researches on data abstraction and distributed computing led to the construction of systems constituted by a collection of "physically" different objects.
The term 'object' was independently adopted by the operating system [Jon], programming language [Gol83] and even by microprocessor manufacturing companies [Int] to denote components having a hidden state and a set of operations or capabilities for transforming the state so as to perform some useful work. Moreover, objects and relations between them were introduced into the artificial intelligence community as basic means for representing knowledge [Min].

Therefore, the object model has been recognized as being able to play a key role in a wide spectrum of applications whatever the style of programming, and suited linguistic constructs have been introduced in languages of each style for expressing it. A partial exception to this rule is logic programming.
The oldest language of this category, i.e. Prolog [Clo] is only ten years old.
It has rapidly increased in popularity not only due to the choice made by the Japanese project [Mot], but also to the increasing demand for knowledge-based [Nau] applications. Prolog is still able to provide intrinsically two forms of modularity:
a) distinction between knowledge base and control;
b) distinction between different facts and rules.
But, at the moment, it is impossible to express in Prolog the third level of modularity and protection required by an object model.
Extensions to Prolog have been proposed in this sense. M-Prolog [Ben] is an example, but the lifetime of modules is here not propagated till execution time. Mandala [Fur] is the first answer of ICOT [Mot] to the challenge: it conceives programs as collections of units characterized by well-identified types of links between them. But Mandala is still a research project, founded

on a different computational model (Concurrent Prolog [Sha]) from Prolog.
As a consequence, in the logic programming community it is still impossible to build a large software system according to an object model; moreover, the intrinsic sequentiality of Prolog prevents the exploitation of multi-processor architectures.

This work has the fundamental aim of introducing a set of concepts and mechanisms to be added to or superimposed on Prolog in order to allow the splitting of a program into a collection of interacting objects. Our approach is based on the fact that the specification of static or dynamic interconnections between objects can be viewed as the specification of a meta program and that meta programming [Aie] is quite a natural job in Prolog. The procedural interpretation of Prolog predicates leads directly to a model of programs as separate worlds - each embedding a piece of knowledge over a particular domain - able to communicate through (remote) procedure calls. Predicates can then be viewed as communication channels to be dynamically connected by meta programs.
Instead of following the approach of amalgamating [Bow] the object-level and the meta-level language we suppose a basic mechanism able to introduce a clear distinction between these two levels, in order to enhance the re-usabilty of objects.
The aim of the paper is to show how easily solutions to classical problems can be obtained and how the concept of meta-programming could be one of the general concepts to use in any style of programming for building programs 'in-the-large'.

1. P-UNITS AS OPEN, LAYERED WORLDS

A logic program is intended here as a collection of objects which will be referred in this work as Prolog-units (P-units) or simply units.
Each P-unit represents a chunk of knowledge about a particular domain and it is conceptually an autonomous world able to solve problems in the form of goals asked of it by users. The users of a unit may be human beings or other units. The answer of a unit to a query consists either in the demonstration of the goal and the consequent binding of variables, or in the notification of a failure. In the second case, this does not necessarily mean that the goal is logically false; it simply means that the unit is not able to perform a valid deduction for it. We will denote such an answer as 'unknown'.

Thanks to the possibility of asking goals of other units, each P-unit can be modeled as an Open World [Hew]. Communication between units is achievec through the predicate:

```
send( Dest, Goal, R_of_G )
```

which succeeds if the unit 'Dest' is able to demonstrate with result R_of_G the goal 'Goal' (represented by a list). The name 'send' recalls that the query could imply, at implementation level, an exchange of messages between the calling unit and the 'Dest' unit when it is allocated on a different processor.
In this sense, send can be interpreted as a synchronous message passing primitive in which the caller always waits for the answer.

A system is conceived as depicted in figure 1. The unit 'system' plays the same role as the 'system package' in Ada (*) [DOD]. It defines system predicates and is conceptually part of every other unit.
The computation starts from the unit 'user'. The user interface provides for the translation of a goal G into the request for a specific unit through a call like:

```
send( dest_unit, G, true ).
```

If the send is demonstrated with success, the answer is given to the user in the current Prolog form. If other solutions are possible, the dest_unit asks the user for 'more?' solutions. If the answer is 'yes', the dest_unit backtracks; otherwise the computation ends.

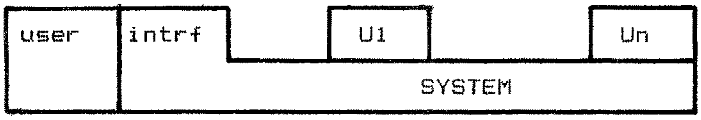

Figure 1

Let us consider, as an example, a system constituted by a unit 'list', which embeds operations on list structures, and a unit 'names', which includes knowledge about alphabetical strings (figure 2). The first P-unit makes reference to the operator 'isless' implemented by the second one in order to perform an ordered insertion of a new name into a list.

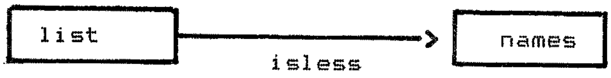

Figure 2

```
insert( [A¦B],C,[A¦D] ):-
   send( names, [isless, A,C ], true ),
   insert( B,C,D ).
insert( [A¦B],C,[C,A¦B] ).
insert( [], C, [C] ).
```

In this case, the query of 'list' to 'names' is equivalent to the static binding of a client unit to its server. It could be claimed that the same result is implicitly expressed through a notation like the 'with' clause of Ada packages [DOD]. An example could be:

```
with( names ).
use( names ).
P-unit( list ).
     insert( [A¦B],C,[A¦D] ):- isless( A,C ), insert( B,C,D ).
     insert( [A¦B],C,[C,A¦B] ).
     insert( [], C, [C] ).
```

where:

(*) Ada is a trademark of the U.S. D.o.D.

P-unit(list) asserts that 'list' is the name of the unit;
with(names) asserts that 'list' sees 'names' as a server;
use(names) asserts that each predicate not defined in 'list' has to be solved by the unit 'names'.

The advantage of such a notation is to introduce clear separation between the phase of programming in-the-small from that of programming in-the-large. For example, an ordered list of integers can be implemented by simply modifying the 'with' and 'use' clauses.
Speaking in terms of a declarative style of programming, the specification of unit interconnection consists in the specification of 'meta' knowledge.
The textual separation of meta knowledge from 'object' knowledge is already a good step towards enhancing the re-usabilty [Weg] of object modules. P-units, however, cannot be simply assimilated to Ada packages: they continue to exist at execution time with an independent lifetime. As a result, the separation between meta and object knowledge can be maintained at run time too, by committing the job of interconnecting P-units to meta-units.
Thus we admit that each P-unit can be associated with a meta-unit and that the basic machine of each unit is able to implement automatic communications between these two layers.
The connection between a client and a server can be dynamically established by the meta unit of the client. This can be achieved by connecting the 'output' channel represented by the predicate to be solved with an 'input' channel of the server, using the send predicate.

1.1 Basic organization of the system.

A P-unit U consists of:
a) a set of Prolog clauses that represents an Object Knowledge Base (OKB) about a particular domain;
b) a Base Machine (BM) or interpreter that is able to answer demonstration requests.

Each external request for the demonstration of a goal implies the creation of a new activation (instance) of the BM, identified by a name defined by the system or by the user (see section 3). This name is a list where the first element is the global identifier of the P-unit and the second a local identifier for the instance.

Let us indicate with G a goal to be demonstrated by U as a consequence of a request from a different P-unit and with SG a subgoal, part of the demonstration process of G.
Before starting the demonstration process for G or SG, the basic machine BMU of U asks its meta-unit MU for:

todemo(Caller, GoalRep, Result)

where 'GoalRep' is bound to an internal representation (a list) of the goal, 'Caller' is bound to the identifier of the calling unit, and 'Result' represents the result of the demonstration of 'Goalrep'. It can be bound to specific values in accordance with the following cases:
a1) todemo succeeds and 'Result' is true. The goal has been demonstrated and some of its output variables may have been bound. BMU goes on. If this is the

case, the results are communicated to the 'Caller';
a2) todemo succeeds and 'Result' is bound to a value different from 'true' (see section 2.1). G is considered as failed and such an event is communicated to the 'Caller'. For SG, BMU has to perform backtracking;
a3) todemo fails: the goal fails at object level too.

The main task of a meta-unit is that of deciding how to solve predicates at the place of its object unit. In order to perform such a task, a meta-unit can:

1) solve the goal directly;
2) ask a different unit for the solution;
3) send a request for the same or a different goal to its own object unit. This case is expressed (for historical and practical reasons) by invocation of the following predicate:

   demo(Goal, Result)

   which can be considered similar to the built-in Prolog predicate "call". A main difference is that demo is always satisfied unless a system crash occurs. Result can be bound to 'true', 'unknown', or to some other value.

Careful implementation is obviously necessary to make such an organization efficient. Several optimizations are possible. The most evident is to avoid dynamic calls to meta-units by performing a static analysis of clauses.

2. PROGRAMMING WITH META-UNITS

2.1. Negation as failure.

Clark's negation-as-failure [Cla77] can be easily expressed:

```
todemo( Caller, [not|X], R ):-
      demo(  X, R1 ), invert( R1,R  ).
invert( true,    false ).
invert( unknown, true  ).
invert( false,   true  ).
```

If X is provable by the object unit, then the result 'false' is explicitly given to the caller. If X is not provable (i.e. the result of the demonstration in the object unit is 'unknown' or 'false'), then the result 'true' is returned. Let us note that 'false' is a new possible result of proof explicitly introduced at the meta-level in accordance with the 'Closed World Assumption' [Cla77].

2.2. System configuration.

A fundamental advantage of using meta programming lies in the re-usability [Weg] of P-units, i.e. the possibility of defining different systems with the same components connected in different ways.
In the case of figure 2, for example, the unit 'list' could be associated with the following meta-unit:

```
meta-unit( list ).
    todemo( C, [isless,A,B], R ):-
          !, send( names, [isless,A,B], R ).
    todemo( C, G, R ):- demo( G, R ).      /* default case */
```

according to the picture of figure 3.

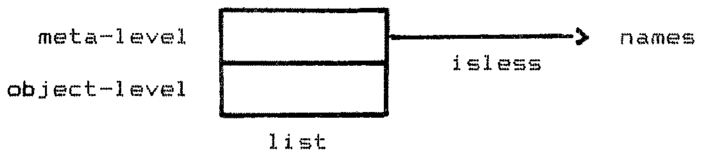

Figure 3

The unit 'list' can specify the 'insert' operation as follows:

```
unit( list ).
     insert( [A¦B],C,[A¦D] ):- isless( A,C ), insert( B,C,D ).
     insert( [A¦B],C,[C,A¦B] ).
     insert( [], C, [C] ).
```

In this specification no configuration policy is embedded in 'list' unit. Consequently, a different system can be defined by simply changing the meta level. This is still an example of static connection between units. It is not however difficult to select connections dynamically. For example:

```
meta-unit( list ).
      todemo( C, [insert¦ARGS], R ):-  !, demo( [insert¦ARGS], R ).
      todemo( C, [OP¦ARGS], R ):-
            are_integers( ARGS ), !, send( integers, [OP¦ARGS], R ).
      todemo( C, [OP¦ARGS], R ):-    send( names, [OP¦ARGS], R ).
      are_integers( A ):- ...
```

Predicates different from 'insert' are solved by the unit 'names' or 'integers' according to the their type of arguments.

2.3. Query the user.

This problem can be considered as another example of dynamic selection of static connections. If we write:

```
meta-unit( list ).
    todemo( C, [isless,A,B], R):-
            send( integers, [are_integers,A,B], true ), !,
            send( integers, [isless,A,B], R ).
    todemo( C, [isless,A,B], R):-
            send( names, [are_names,A,B], true ),!,
            send( names, [isless,A,B], R).
    todemo( C, [isless,A,B], R):-
            send( user, [question,isless,A,B], R ).
    todemo( C, G, R ) :- demo( G, R ).  /* default case */
```

then the solution of the 'isless' predicate is asked of the user when the arguments belong to types which the system does not know, i.e. when they are not integers or names.

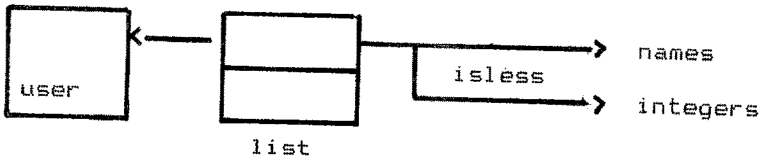

Figure 4

2.4. Type and right checking and exceptions.

Calls like:

```
send( integers, [are_integers,A,B], true ),  or:
send( names, [are_names,A,B], true ),
```

in the previous example may be interpreted as dynamic type checks on data structures, to be performed by the unit which represents the "data type".
In a perfectly similar way it is possible to check the caller's access rights.
For example:

```
todemo( Caller, [OP¦ARGS], R ):-
    send( user-rights, [check, Caller, OP], true ), !, demo( [OP¦ARGS], R).
todemo( Caller, G, failure ):-
    send( exception-unit, [access_denied, Caller,G], true).
```

avoids the execution of an operation when the caller has no right to it. Knowledge of access rights is here supposed to be embedded in the 'user-rights' unit. If the Caller has no right to the operation OP, the call to user_rights fails and the error is recorded in an 'exception-unit'. The result of the proof is set to 'failure'. The meta-level can interpret such a value as an excpetion to be handled.

2.5. Multi-layered systems.

A meta-unit MU is, in its turn, a P-unit. This means that it can be associated with a meta-unit MMU and so on. Then a system can be conceptually split into a hierarchy of operationally interlaced layers.
As an example, let us consider a debugging session of the system of figure 4. The tracing of the behaviour of the unit 'list' can be implemented by adding a second layer within 'list':

```
meta-meta-unit( list ).
        todemo( C, [todemo, Caller, G, R_of_G], R ):-
                send( debug,[tracegoal, G ], true ),
                demo( [todemo, Caller, G, R_of_G], R ),
                send( debug, [traceresults, G, R_of_G ], true ).
```

The unit 'debug' provides for printing each goal of 'list' before and after its demonstration process.

2.6. Knowledge-based systems.

The following specification:

```
    todemo( Caller, G, R_of_G):-
            my_name( MYSELF ), find_super( MYSELF, SU ),
            send( SU, G, R_of_G ).
```

provides for routing G to the unit SU, which is qualified as the 'super' unit of the current one (identified by the system predicate 'my_name'). An interpretation of super units may be related to the concept of inheritance of properties, which is characteristic of knowledge representation systems [Dav]. In this case, G is part of the external interface of the module represented by the current unit, but its "body" is defined within SU.

Meta_units can be introduced to express and control default knowledge. An example of a system able to express that "Penguins are birds, but they do not fly" may be:

```
meta-unit ( penguins ).
   /* any attempt to demonstrate that penguins fly has to fail*/
   todemo(C, [fly], false) :- !.
   /* any other property has to be deduced from birds */
   todemo( C ,G ,R_of_G ) :- send( birds, G, R_of_G ).
```

3. CONCURRENT SYSTEMS AND SHARED UNITS

If P-units do not have an internal state then the same system can be accessed in parallel by a multiplicity of users without any side effect. In Prolog the concept of state is often related to the set of clauses which constitute a program and which can be modified through built-in predicates like 'assert', 'retract' etc. If such a modification is admitted for P-units also, then concurrent accesses must be disciplined through suited mechanisms for mutual exclusion and synchronization.

The solution given to this problem in the context of traditional, imperative programming languages is well known. Linguistic constructs such as semaphores or monitors [Hoa] allow us to solve mutual exclusion and synchronization problems according to a 'global environment' model of programming [And].

This model could be adopted for P-units too. Special units could be introduced, working as semaphores or monitors, in order to ensure a disciplined modification of internal states.

But P-units are not passive resources. A P-unit is made of a collection of different, internal processes (which have been called instances) each activated by a specific external request. If each instance is made able to perform an explicit control on communications with its clients, then it can be perceived from outside as an object able to control modifications its internal state: the most appropriate model for communication and synchronization is then the 'local environment' model [And].

The advantage of this approach is not only that it is suitable for distribution of units into different nodes of a network, but also that it allows the representation of state through logical variables.

3.1. Process communication and synchronization.

A process (instance) controls the processing sequence of external requests through synchronization clauses of the form:

```
entry(...), accept(...) :- body( ... ).
```

A synchronization clause is based on a concept very similar to that of 'event' in Distributed Logic [Mon] and in Delta Prolog [Per]. 'Entry' represents an "interface predicate" : it is a predicate which can be called by other units through the usual send primitive and can appear at the head of a synchronization clause only. If a unit C calls:

```
send( [U,P], [entry|ARGS], Res )
```

then the demonstration of 'entry' is asked of the particular instance [U,P] (let us note that, if P is not bound, then the request is sent to any instance of U). As usual, a new instance [U,E] is created by U to perform such a demonstration. This instance, however, must cooperate with [U,P] according to the following rules:

r1) the 'entry' request must unify with the left part of the head. If no unification is possible, [U,E] terminates with answer 'unknown'. If unification is possible, [U,E] is 'frozen' until [U,P] completes a successful unification of the 'accept' part of the head;

r2) if [U,P] tries to unify with 'accept' when no external request exists at all, then it is suspended. If there are external requests for [U,P] but different from 'entry', then [U,P] backtracks. If more than one request for 'entry' exists, then the first is selected;

r3) when both parts of a synchronization clause have been jointly unified with success by [U,E] and [U,P], then the 'body' of the synchronization clause is jointly executed by the two instances. If the joint unification or the 'body' fails, [U,P] and [U,E] backtrack. Therefore, if no further synchronization alternative is available for 'entry', [U,E] terminates with answer 'unknown'. If the 'body' completes with success, [U,E] completes with answer 'true'. No undoing and backtracking is admitted for entry predicates.

According to the previous rules,the failure of a query by C to [U,P] means that [U,P] is not able to solve the problem 'at this moment'. But it could have success 'in the future', in correspondence with the specific value of its internal state. Let us consider the example of a buffer:

```
unit( buffer ).
       (1) buffer(B):- notempty(B),  remove(B,C),    buffer(C).
       (2) buffer(B):- notfull(B),   insert(X,B,C),  buffer(C).
       (3) buffer(B):- buffer(B).
       (4) get(X), remove(B,C):-  out( X,B,C ).
       (5) put(X), insert(B,C):-  in( X,B,C ).
       (6) init :- buffer([]).
       (7) ?- init.
```

Predicates 'get' and 'put' constitute the buffer interface. The process [buffer,_] is an end-less loop which starts in an autonomous way (7). It has

two synchronization points with the external world: (4) and (5). Predicates 'notempty' and 'notfull' play the role of local guards to control the non-determinism of communications. The clause (3) is selected when the first two fail; at implementation level, the control of a shared processor could be released at this point and memory resources can be saved according to tail recursion optimization schemes. The state of the buffer is represented by the variable B.
If a producer process calls 'put' when the buffer is full, its request fails. But the meta-unit of buffer (or the producer itself) can resend the request until it is satisfied:

```
meta-unit( buffer ).
  todemo( C, [put,X], true):-
        repeat, demo( [put,X], true).
meta-unit( producer ).
  todemo(X, [deliver¦Z], Res):-
              send( [buffer,_], [put¦Z], Res).
unit( producer ).
    prod:- read(CH), deliver(CH), prod.
```

Obviously, such a policy could be frozen by changing the basic semantics of synchronization clauses. Rule r3) could become:
r3) ... If no further synchronization alternative is available for entry, then
[U,E] is placed at the end of the request queue. ...
The advantage could be that of avoiding starvation. The model of processes intraction is in this case very similar to Ada's rendez-vous.

3.2. Asynchronous activation of processes.

Till this moment no mechanism exists for creating new processes without being obliged to wait for their termination. To overcome such a drawback, the system predicate:

start(U, P, IG)

is introduced. It creates a new instance for the P-unit U, with name [U,P] and initial goal IG. It is satisfied at the end of the activation phase, i.e. it has not need to wait till the end of [U,P]. A simple fork operator can thus be implemented as follows:

```
meta-unit( u ).
     todemo( C, [fork,NAME], true ):- !,start( u, NAME, init ).
     todemo( C, G, R ):- demo( G, R ).
unit( u ).
     p1( ... ):- ..., fork( u1 ), ..., p2(...).
     p2( ... ):- ....
     init :-      p1( ... ).
     ?- init.
```

At the very beginning, the system is constituted by the instance [u,-] only. At the end of 'fork' this instance runs in parallel with another [u,ul].

A more complete form of 'start' could provide an additional parameter to specify a 'continuation' unit which, as in languages based on the Actor model [Fil], could collect results of a process when it terminates. An example of this will be shown in section 3.4.

3.3. System configuration.

Starting from the same units as the previous producer-consumer system, we can now configure different systems, by simply changing the initialization phase and the meta layer. The initialization of buffer could become:

```
     (6)  init( X ):- buffer( X ).
```

and the meta layer of producer:

```
meta-unit( producer ).
  todemo( [producer,prod1], [deliver,Z], R ):- send([buffer,buf1],[put,Z],R ).
  todemo( [producer,prod2], [deliver,Z], R ):- send([buffer,buf2],[put,Z],R ).
```

If we write the following unit:

```
unit( system1 ).
    init_system:-
           start( producer,  prod1,  prod ),
           start( producer,  prod2,  prod ),
           start( buffer,    buf1,   init( [] ) ),
           start( buffer,    buf2,   init( ['i'] ) ),
           start( consumer,  cons1,  cons ).
?- init_system.
```

then we create a system made of two producers, (prod1 and prod2), each connected to a different buffer (buf1 and buf2 respectively), and a single consumer. Buf1 starts as an empty buffer, whereas buf2 has initial contents.

According to the discussion of section 2, dynamic connection of interprocess communication channels is possible. A particular unit could, for example, be introduced to maintain knowledge about connections and create new ones. A system which assists the user in expressing his own software systems is usually referred to as a programming environment. In this context, the main job of a programming environment could then be that of 'writing' the meta-layer of P-units and processes according to the user's configuration commands.

3.4. A shell process.

The shell is a typical tool of advanced programming environments. A shell process could be implemented through P-units in several ways. One of this is:

```
unit( commands ).            unit( shell ).
    dir(   ... ).            go :- accept, go.
    print( ... ).            go :- go.
    mkdir( ... ).             command([bg|X]), accept :-
    ...                            start( commands, _, X, result_unit ).
                               command([fg|X]), accept :-
                                   send(  commands, X, R , display( X, R ).
                               display( G, R_of_G ):- ...
                               ?- go.
```

When a user gives a command (say print in background), the user interface provides to send the command to the shell process:

```
send( [shell,_], [command,[bg,print,X] ], R).
```

To serve a background request the shell activates a new 'commands' process using the extended form of 'start' primitive. The answer to the command is stored, when this new activation terminates, in the 'result_unit' for later consultation by the user. A foreground request is immediately served by sending the request to the 'commands'. The shell process waits in this case for execution of the command to end. Let us note that the possible failure of a background or foreground command does not cause (as happens in other approaches such as Concurrent Prolog [Sha]) the failure of the shell process since the success of 'start' and 'send' is not related to successful termination of the particular user command. Besides, the output generated by the foreground command is immediately visible to the user. This kind of solution is more oriented towards an incremental and modular design of applications than those proposed in Concurrent Prolog [Sha] or PARLOG [Cla84].

4. CONCLUSIONS

In this work we have explored the possibility of expressing programs as collections of Prolog objects, so as to allow the splitting of a single base of clauses into a set of dynamically recognizable modules (P-units). The main motivation of the work was to introduce in the context of the more widespread logic programming language, a set of concepts whose validity has already been ascertained in other styles of programming.
The most interesting result of this attempt lies perhaps in the new, great expressive richness that has been achieved and which consists mainly in the number of different possible interpretations for P-units.

A P-unit has been introduced as a single object able to solve problems in a particular domain. The clauses of a P-unit were then considered as the static description of the behaviours of objects (instances) dynamically created to solve external requests. In this sense, a P-unit can be viewed as a 'class' whose basic specification can be modified or integrated by that of the associated meta-unit(s). Each instance can be interpreted as a process which interacts with other instances through a rendez-vous paradigm. The interconnection between processes can be statically or dynamically specified at meta-level with the advantage of avoiding any change in the object level. From the meta-level point of view, predicates can be considered as input-output channels to be connected together to achieve the desired communication. References to units can be conceived as capabilities to be checked and handled in a transparent way with the respect to the object level.

Explicit synchronization between processes has been achieved through synchronization clauses, chosen for their well-established semantics [Mon] and their uniformity with the rendez-vous paradigm. The kind of parallelism we have expressed is different from that of other proposals such as Concurrent Prolog [Sha] or Parlog [Cla84], which introduce "don't care" non-determinism and guarded clauses to exploit the potential parallelism which is intrinsic in clauses. Interprocess communication occurs in these languages through streams, a fact which, in our opinion, makes these proposals suited for programming units (P-units too) in-the-small, but rather unsuitable for expressing interactions in-the-large.

In summary, P-units can be viewed as objects, classes, passive resources or as processes. The network of their static relationships can be conceived as the representation of knowledge on the applicative domain, whereas their dynamic relationships could be viewed as the dynamic (re)organization of such knowledge according to events. Systems like Mandala [Fur] can then be conceptually implementable in terms of P-units.

Finally, the proposed paradigm can be used both for application and system design. This means that knowledge can be easily shared and that extensible systems can be easily built. In particular, P-units seem well suited to build a programming environment for Prolog, using Prolog itself. The advantages of using the same language for designing and building its own programming enviroment has been learned from systems like Interlisp [Tei] or Smalltalk [Gol84]. We believe that these advantages could be fundamental in the logical programming community too. This is one of the goals we intend to pursue in the future. At the moment a prototype implementation of the proposal has been written in Prolog on a personal computer and an implementation based on modifications of an existing Prolog interpreter is in progress on a Sun machine.

REFERENCES:

- L.Aiello, G.Levi:" The uses of meta-knowledge in AI Systems", ECAI-84, Pisa, Sept. 1984.
- G.R.Andrewes, F.B.Schneider: "Concepts and Notations for Concurrent Programming", ACM Computing Surveys v.15,n.1, March 1983.
- J.Bendl, P.Koves, P.Szeredi :" The MPROLOG System", Proceedings of the Logic Programming Workshop, july 1982.
- K.Bowen, R.Kowalski: "Amalgamating language and metalanguage in logic programming", in Logic Programming , Academic Press, 1982.
- K.L.Clark :"Negation as failure", in "Logic and Databases", Gallaire and Minker eds., Nov.77.
- K.Clark, S.Gregory :" PARLOG: Parallel Programming in Logic", Research Report DOC 84/4 , Imperial College, 1984.
- W.F. Clocksin, C.S. Mellish : " Programming in Prolog ", Springer-Verlag, New-York, 1981.
- R.Davis, D.Lenat :" Knowledge-Based Systems in Artificial Intelligence ", New York: McGraw-Hill, 1980.
- DOD: "Reference Manual for the Ada programming language", ANSI/MIL-std 1815-a, Jan.1983.
- R.E. Filman, D.P. Friedman: "Actors", in 'Coordinated Computing', Prentice-Hall, 1984.
- K.Furukawa et alii: "Mandala: A Logic Based Knowledge Programming System", in International Conference On Fifth Generation Computer Systems 1984.

- A. Goldberg: " SMALLTALK-80: the Interactive Programming Environment", Addison-Wesley, 1984.
- A.Goldberg, D. Robson: "Smalltalk-80, The Language and its Implementation", Addison Wesley, 1983.
- C.Hewitt, P.De Jong :"Open Systems", Tech. Rep. MIT-AIM 691 Dec. 1981.
- C.A.R Hoare: "Monitors: An Operating Systems Structuring Concept", Comm. of ACM, 17 (10): 549-557, 1974.
- Intel:"Introduction to the iAPX-432 Architecture", Intel on. 171821.
- A.K.Jones: "The Object Model: a Conceptual Tool for Structuring Software", in 'Operating Systems', ed. by Bayes et al., Springer Verlag, n.60, 1978.
- M.Minsky: "A Framework for Representing Knowledge", in "The Psychology of Computer Vision" (ed. P.Winston), Mc Graw-Hill, 1975.
- L.Monteiro :" A Proposal for Distributed Programming in Logic", Tec. Rep. University of Lisbona, Jan. 1983.
- T. Moto-oka et al.: "Challenge for Knowledge Information Processing Systems (Preliminary Report on Fifth Generation Computer Systems)", Proc. of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, October 1981.
- D.Nau: "Expert Computer Systems", Computer, v.16, n.2 Feb. 1983.
- L.M.Pereira, R.Nasr:"Delta-Prolog: A Distributed Logic Programming Language", International Conference On Fifth Generation Computer Systems, November 1984.
- E.Y. Shapiro: "A subset of Concurrent PROLOG and its Interpreter" Technical Report ICOT n.3 , Oct. '83.
- W. Teitelman, L.Masinter :" The INTERLISP Programming Environment", IEEE Computer, v.14, n.4, 1981.
- P. Wegner :"Capital Intensive Software Technology" , IEEE Software , v.1, n.3, 1984.