PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

MODULAR AND COMMUNICATING OBJECTS IN SICSTUS PROLOG

Nabiel A. Elshiewy

ELLEMTEL Telecommunications Systems Laboratories*
Box 1505, S-125 25 Älvsjö, Sweden

ABSTRACT

An experiment to integrate the object-oriented paradigm of programming into SICStus Prolog is reported. This integration supports the capabilities of encapsulation and organisation of large-scale logic programs, for product software development, using hierarchical structures and encapsulated inheritance mechanisms and making distinction between code inheritance and behaviour inheritance. Making use of SICStus Prolog facilities to delay the evaluation of calls waiting for variable bindings, systems can be described in terms of asynchronously communicating objects with private memory and internal states changeable in response to external messages. Buffering and arbitration of incoming messages are realized by stream communication and merge with constant delay.

1 INTRODUCTION

Many product software applications are complex and large and are often composed of different components and program modules. To allow logic programming of such applications, intelligent mechanisms and tools are required to support modular design of their components (modules) and also to support modification, reuse and interconnecting of the component modules.

In the object-oriented paradigm of programming, systems are described in terms of computational entities called objects which are interacting by sending messages to each other. Each object encapsulates its own private memory and computational behaviour. An object is an instance of a class which groups all similar objects and defines their structure. Classes can be organized into hierarchies. An object belonging to a class in the hierarchy inherits attributes of all objects on the higher level.

This paper presents an object-oriented logic model integrated with and implemented in SICStus Prolog [Carlsson and Widen 1988]. This integration supports capabilities for organizing large-scale logic programs by the use of hierarchical structures and inheritance mechanisms. The object-oriented message passing paradigm also allows logic programming of many systems which are naturally described in terms of asynchronously communicating objects with private memory and internal states changeable in response to external messages. The integrated system provides a stronger support for encapsulation which is an important requirement of product software development.

* This work was done while the author was visiting at the Swedish Institute of Computer Science, SICS.

Many attempts to integrate the object-oriented paradigm into logic programming have been reported using either Prolog, e.g. [Chikayama 1984, Zaniolo 1984, Gullichsen 1985] or parallel logic languages, e.g., [Shapiro and Takeuchi 1983, Kahn et.al 1987, Yoshida and Chikayama 1987, Davison 1988]. Proposals extending Prolog either failed to support the encapsulation of state changes or made severe restrictions on the use of the concepts of logic programming.

The object-oriented style of programming using parallel logic languages, first explored in [Shapiro and Takeuchi 1983], provides encapsulation of objects by describing objects in terms of tail-recursive processes. The state of the object is given as a set of arguments to the process which may change for each recursive call to the procedure. Messages are communicated between objects through partially instantiated shared variables (streams). An object is suspended until it receives a message. A simple inheritance mechanism is provided allowing an unrecognized message to be passed over a stream connecting the receiving object to the inherited object (explicit delegation to parts).

It has been argued [Kahn and Miller 1987] that a Prolog system which provides either a wait declaration or a "freeze" primitive can support object-oriented programming in a manner similar to parallel logic languages. SICStus Prolog provides both a wait declaration facility and a "freeze" primitive in addition to other primitives useful to support object-oriented programming [Carlsson 1987].

The approach presented here is similar to the one supported by parallel logic languages with the main difference that it supports full encapsulation [Snyder 1986] which means that not only instance variables and internal behaviour are hidden from the clients of a class but also the inheritance hierarchy it might make use of. Only the methods of a class may be inherited. Full encapsulation provides greater support for a strong program modularity, in particular, for reusability and dynamic modification (redefinition) of program modules.

In this paper, an overview of the integrated system is given describing its major features illustrated with examples. The implementation strategy is described and shown to be simple and efficient with a minimum of overhead.

2. CLASSES AND INSTANCE BEHAVIOUR

To preserve the full benefits of encapsulation, a class defines an abstract external interface which only contains

the set of messages accepted by instances of that class. The abstract external interface serves as a contract between the object and its clients. Each object is an instance of one class and the object may receive messages from a multiple of dynamically created client objects. An object is known to each of its clients by a mail box (stream) associated with it on which the clients may send contracted messages. Multiple streams are fair merged into the server object.

A message has a name and (possibly) a collection of arguments. Making use of the logical variable concept, some of the arguments of a message may be variables to be bound by the object receiving the message which is referred to as a reply from the object to its client. The class definition contains the appropriate code which describes the computations to be performed in response to the messages received by each object.

A new class is defined by its name and the abstract external interface to its instances. In the present implementation it is syntactically described using the following declaration:

```
:- class(ClassName, ListOfMessages, InitCmds).
```

ListOfMessages defines the protocol of messages to be accepted by the instances. A class declaration does not provide any further information about instance variables or use of inheritance. Such information is fully encapsulated in the class and its instances. It may, however, be required to allow creating an object with different initial data prompted by the creator of the object. This is provided by the abstract interface declaration of the class in the third argument *InitCmds* which defines the set of commands to be supplied by the creator of the object to enable alternative initial states for the created object.

A simple example of a class definition for a queue of elements is given below:

```
:- class(queue, [enqueue/1, dequeue/1, is_empty/1,
                 length/1, display/1], []).
```

which means that an object of *queue* may accept five messages all with one argument and no initialization commands are provided. If it is permitted to define an arbitrary initial state when creating a new instance of the class *queue*, the designer of the class may provide the following abstract external interface:

```
:- class(queue,[enqueue/1, dequeue/1, is_empty/1,
                length/1, display/1], [with/1]).
```

where *with/1* may represent a command with an argument which contains a list of elements to be queued as an initial state of the queue instance. The designer of a class must provide a definition of the procedure *initialize/2* to specify the initial state of class instances. The first argument of *initialize/2* is the list of initialization commands which can be empty and the second argument is the initial state (variables and data structures) of the object. To define the initial state of a queue object, *initialize/2* may have the following definition:

```
initialize([], [q(0, Q, Q)]).
initialize([with(L)], [InitQ]) :-
        mk_queue(L, q(0, Q, Q), InitQ).

mk_queue([], Q, Q).
mk_queue([E|Es], q(N, F, [E|R]), Q) :-
        N1 is N+1,
        mk_queue(Es, q(N1, F, R), Q).
```

The initialization procedure defines the the structure of the initial state and its value is either defined by default (no initialization commands are entered) or according to some given initialization commands. In our example, the state of a queue instance is represented by a difference-list structure which is empty and of length '0' by default. Responding to an initialization command *with/1*, a call to *mk_queue/3* will result in a queue where the elements given as an argument to the command *with/1* are enqueued.

The computations required to respond to messages are defined in terms of calls to local and/or universal procedures. Universal procedures are either built-in or library defined procedures (to be described later). A queue object may behave according to the following (less-verbose) code:

```
queue(q(N, Qh, [E|Qt]) :-
        enqueue(E)  =>      N1 is N +1,
                            queue(q(N1, Qh, Qt)).
queue(q(N, [E|Qh], Qt) :-
        dequeue(E)  =>      N1 is N -1,
                            queue(q(N1, Qh, Qt)).
queue(Q) :-
        (   is_empty(yes)   => Q = q(0, _, _)
        ;   is_empty(no)    => Q = q(N, _, _), N > 0
        ;   length(N)       => Q = q(N, _, _)
        ;   display(L)      => portray(Q, L)
        ),
        queue(Q).

portray(q(0, Q, Q), []).
portray(q(N, [E|Qh], Qt), [E|L]) :-
        N1 is N-1, portray(q(N1, Qh, Qt), L).
```

In the above program, the communication streams are abstracted away. Only the state of the object is given as an argument to the main procedure describing the behaviour of queue instances. The main (message dispatching) procedure has the same name as the class defining it. The infix operator '=>/2' is introduced of which the left-hand argument is a message and the right-hand argument is the computations performed in response to the message.

To flag the end of a class definition, the following declaration is used:

```
:- endclass(ClassName).
```

This is translated to the following SICStus Prolog program in which two extra arguments are added to the head and the recursive calls of *queue/1* above, the first is the input stream on which messages are received from outside and the second is the *SelfStream* on which messages to *Self* are sent. To enable a queue object to be activated (waken-up) only when a message arrives on its input stream, SICStus Prolog's wait declaration is used.

```
:- wait queue/3.
```

To evaluate a call to a procedure, with a *wait declaration*, the first argument of the call has to be

instantiated (bounded to a non-variable term). The call reduction is, otherwise, delayed until the argument is bound to a non-variable term possibly by another call in a conjunction of calls.

```
queue([enqueue(E)|CS], SS, q(N, Qh, [E|Qt])) :-
        N1 is N+1, queue(CS, SS, q(N1, Qh, Qt)).
queue([dequeue(E)|CS], SS, q(N, [E|Qh], Qt)) :-
        N1 is N-1, queue(CS, SS, q(N1, Qh, Qt)).
queue([is_empty(yes)|CS], SS, Q) :-
        Q = q(0, _, _), queue(CS, SS, Q).
queue([is_empty(no)|CS], SS, Q) :-
        Q = q(N, _, _), N > 0, queue(CS, SS, Q).
queue([length(N)|CS], SS, Q) :-
        Q = q(N, _, _), queue(CS, SS, Q).
queue([display(List)|CS], SS, Q) :-
        portray(Q, List), queue(CS, SS, Q).
queue([], [], Q).
```

The designer of a class has the freedom to choose the names of the local procedures in the class definition. To avoid multiple name conflicts, each local procedure name is prefixed by the name of its class attached to it by the operator ':/2' when translated and loaded into the system. In the example given above, the local procedures are named: *queue:initialize/2*, *queue:mk_queue/3* and *queue:portray/2*.

The fact that a class defines a fixed set of messages led to the decision that if an undefined message is received by an object (message unification failed), the result of the computation is failure. No switching to a default method or handler is provided. Handling exceptions and incomplete knowledge is, however, a subject under study.

3 MAKING INSTANCES

To avoid the need for meta-class hierarchies, an instance of a class is created by a call to the (system-defined) procedure *make_instance/3* which behaves according to the following program:

```
make_instance(CN, EMS, InitCmds) :-
        make_instance(CN, EMS, InitCmds, SelfS).
```

The call takes three arguments: *CN*, the name of the class to which the instance belongs, *EMS*, the external stream on which messages are sent to the instance and *InitCmds*, a list of initialization commands (possibly empty). Identifying a self stream *SelfS*, *make_instance/3* is reduced to a call to *make_instance/4*:

```
make_instance(CN,EMS, InitCmds, SelfS) :-
        ensure_loaded(class(CN)),
        merge(EMS, InputS),
        priority_merge(SelfS,InputS,  IMS),
        ancestors(CN, As),
        form_instances(As,CN,IMS,SelfS,InitCmds).

form_instances([], CN, IMS, SelfS, InitCmds) :-
        initialize(CN, InitCmds, InitState),
        Instance =.. [CN, IMS, SelfS| InitState),
        call(Instance).
form_instances(IAs, CN, IMS, SS, InitC) :-
        inherit_instance(IAs, InitC, SS, DS, InhIns),
        initialize(CN, InitC, InitState),
        append(DS, InitState, TArgs),
        Ins =.. [CN, IMS, SS| TArgs),
        activate([Ins|InhIns]).
```

The definition of the class with *CN* is loaded (if not already there).

The ExternalMessageStream *EMS* is defined as a *split stream* [Brand and Haridi 1988] which allows the instance to receive messages from a multiple of (dynamically created) client objects. A split stream *S* is a data structure with the following selectors: *{}*, *[H|S1]* and *split({S1, ..., Sn})*, where *S1*, ..., *Sn* are split streams and *H* is a term). To distinguish between split streams and conventional streams which are represented as a list of elements, the latter will be called *single stream* in the sequel.

Calling the system-defined operator *merge/2*, messages received on the split stream *EMS* are merged into the single *InputS* stream The operator *merge/2* is implemented so that it guarantees constant time access using a destructive assignment primitive (provided in SICStus Prolog) in a similar manner to that given in [Shapiro and Safra 1986].

Each instance possesses its own *SelfStream* on which the object itself or any of its ancestors may send messages to *Self*. The system-defined operator *priority_merge/3* is called to merge both the *SelfS* and the *InputS* streams into an InternalMessageStream *IMS* on which messages from *SelfS* stream are given priority to appear.

The goal *ancestors/2* is evaluated to collect the names of immediate ancestors of the class. An empty list of names is produced if no inheritance is defined. In such case, calling the goal *form_instances/5*, the initial state of the class instance is defined by evaluating the goal *CN:initialize/2* using the code given in the class definition. The goal representing the object is then constructed having the arguments: *IMS*, *SelfS* and the *InitialState* of the object. A possible scenario for the creation and manipulation of a queue object may be as follows:

```
make_instance(queue, CS, []),
CS = [enqueue(1), is_empty(R), dequeue(E),
      length(N), enqueue(2) |CS1].
```

If the class inherits the behaviour of single or multiple immediate ancestors, the goal *form_instances/5* is evaluated first to build up an inheritance tree of instances of all ancestor classes and then to activate the resulting instances as will be shown in Section 5.

4 DYNAMIC ENCAPSULATED INHERITANCE

One of the most important features of object-oriented programming is the concept of sharing knowledge between related groups of objects. An important observation is that sharing between objects is used for two distinct purposes: either to only share code to be reused by different classes with no reference to an external behaviour; or to share both the code and the behaviour (state variables and data structures) of different classes.

4.1 Code Inheritance

If the primary use of inheritance is motivated only by code reuse, the concept of universal procedures (methods)

is provided. A *universal procedure* is either a *built-in* procedure or a *library* procedure. A *library* groups a collection of procedures in code form only. It is different from a class in that a library has no behaviour in the form of instance variables or local state and that no instances can be created for a library.

No special declaration is required for the use of built-in procedure calls. The library system is, mainly, based on the module system provided in Quintus-Prolog version 2.0 [Quintus 1987]. A library module definition begins with a declaration of the form:

```
:- library(NameOfLibrary, PublicPredList).
```

PublicPredList is a list of predicate specifications of the form *Name/Arity*.

If some library defined procedures are used by a class, a directive in the following form is required in the class body:

```
:- use_library(NameOfLibrary).
```

When an instance of a class, which contains this directive, is created, all the procedures stored in the library named *NameOfLibrary* are loaded, if not already there. If a restricted number of procedures in the library is required, the following directive may be used:

```
:- use_library(NameOfLibrary, ProceduresList).
```

where *ProceduresList* is a sequence of one or more elements of the form *ProcedureName/Arity*. For example, if from a library named *arithmetic* which contains procedures for manipulating arithmetic operations, only the procedures *square/2* and *divisible/3* are required, the declaration (directive):

```
:- use_library(arithmetic, [square/2, divisible/3]).
```

is entered according to which the code of the two procedures is loaded from the library *arithmetic* if not already loaded before. The code from a library is loaded only once even if it is used by different classes in the running system.

To avoid name conflicts, the same convention used for local class procedures is used here, namely, prefixing the name of the library attached to the name of the procedure by the infix operator ':/2'.

4.2 Inheriting Behaviour

The mechanism, adopted here, to share semantic knowledge (including both behaviour and code) is based on the concept of *encapsulated inheritance* [Snyder 1987]. Each object is an instance of a class. If a class is defined to share semantic knowledge with an ancestor class, an instance of the ancestor class is created and its input stream is made known to the descendant object. Any message which cannot be processed by the receiving descendant object is delegated to the ancestor object through the *DelegateStream* connected between the two objects. Externally, all objects in the inheritance hierarchy are viewed as an indivisible object. All access to inherited behaviour of objects is mediated by the child object.

The following declaration is required in the definition of a class if inheritance of the behaviour and the code of an ancestor class is required:

```
:- inherits(Ancestor, InhMessages, InhInitCmds).
```

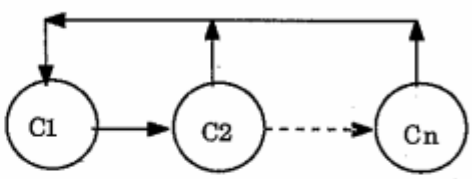
The designer of a class is allowed to choose a subset of the abstract external interface defined for the ancestor class. The second and the third arguments are set to *'all'* if all messages and initialization commands are included in the inheritance. If, otherwise, a subset of the messages and/or the commands are required, they must be defined explicitly.

Multiple inheritance is defined by multiple declarations of the form above. The order of multiple ancestor classes is irrelevant. Messages are delegated to the appropriate ancestor because of the advance knowledge of the ancestors' external message interface.

5 TRANSFORMING THE INHERITANCE TREE

A call to *inherit_instance/5* (defined in Section 3 above) produces instances for all inherited classes in the hierarchy, all with a common *SelfStream* and connected to each other through single *DelegateStreams*. The inheritance tree is represented as a conjunction of calls to the different (dispatcher) procedures representing the different objects in the inheritance hierarchy. The conjunction of calls is viewed, externally, as a single indivisible object. All interaction with external clients is handled by the child instance.

In the case of using single inheritance, i.e., each object has at most one ancestor, the inheritance tree is shown in the figure below.



Making use of the known abstract interfaces, delegation to ancestor instances is generated, automatically, by the pre-compiler. There may, however, be need to explicitly delegate a message to the immediate ancestor class, e.g., processing a message by a descendant requires sending a message to the immediate ancestor to perform a sub-computation. The system-defined primitive *'super/1'* is provided for this purpose. A call *super(Message)* adds the argument *Message* to the *DelegateStream* connected to the immediate ancestor object.

Using multiple inheritance may give rise to the following two problems:

a) a class may inherit messages and/or initialization commands with the same *Name/arity* from more than one ancestor class.

b) the inheritance tree may contain multiple instances of the same class, i.e., two or more classes, higher in the

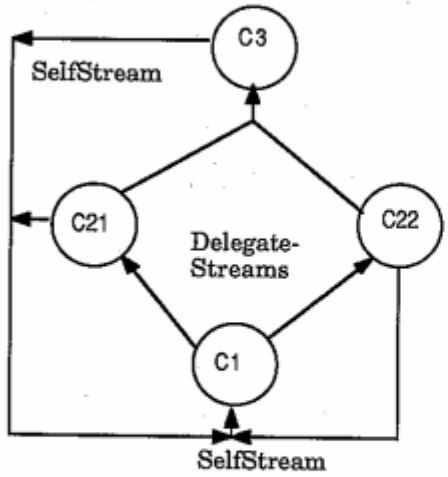
hierarchy, may inherit the same external interface of a common ancestor class.

If several messages and/or initialization commands with the same *Name/Arity* appear in the different external interfaces of multiple ancestors, it is the responsibility of the inheriting class designer to resolve any conflicts. The designer may rename the messages in the external interface of the descendant class or exclude the undesirable messages from the inherited external interfaces of the ancestor classes. Another flexible approach is to use the system-defined primitive '*super/2*' to explicitly delegate a message to a specific immediate ancestor. A call

super(ImmediateAncestor, DelegatedMessage)

adds the *DelegatedMessage* to the *DelegateStream* connected to the instance of the given *ImmediateAncestor*.

One of the reasons for the possible appearance of multiple *Name/Arity* in the external interface of a descendant class is the existence of multiple instances of a common ancestor class in the inheritance tree. Another related observation is that messages processed by procedures defined in the common ancestor class must be delegated to each instance of the common ancestor. To avoid these problems, multiple instances of a common ancestor class are replaced by one single instance. This solution also avoids duplication of instance variables and the related problem of keeping them updated. To allow delegating messages from the different descendant instances, the delegation streams connected to them are merged into one connected to the new common instance. Assume, for example, that a class C1 is defined in terms of inheritance from two classes C21 and C22 and both ancestors are defined so that each of them inherits from the same class C3. The resulting inheritance tree contains two instances of the common class C3. This tree is transformed, further, to the modified tree shown in the figure below.



Note that each descendant object still possess its own identification and view of the (common) ancestor object keeping the delegation stream connecting them unchanged.

5.1 Descendants are Privileged Clients

An important observation related to multiple inheritance and the case shown in the figure above is that if during the processing of a message in the descendant object C1, the message may be delegated to each of the two ancestor objects C21 and C22, respectively. Processing the responses in each object may include sending the same message to their common ancestor C3. This leads to redundant computations in the common ancestor's object. This property is highly undesirable, in particular, if the computations have any side effects.

To avoid this problem, a general design principle is defined in [Carnese 1984] as follows: "*Code which may be useful for methods of descendant* [objects] *should not be defined in procedures which invoke methods of ancestor* [objects]".

The solution adopted here, to solve this problem preserving the full encapsulation property and not exposing the inheritance, is that the designer of a class may consider that the class can be inherited by other classes. Descendant classes may be allowed the privilege to greater access to the internals of the class definition, e.g., access to the direct evaluation of local procedures in the ancestor class. The designer of a class may, therefore, provide a set of messages to be accepted and processed only if sent by a descendant instance. The set of messages cannot be a part of the external interface of the descendant class and cannot be inherited further to any descendants of the descendant. The following declaration in a class definition describes the protocol of privileged messages:

```
:- descends(ListOfPrivilegedMessages).
```

where each member of *ListOfPrivilegedMessages* is of the form *Name/Arity*.

This solution allows the designer of, e.g., the class C21 to factor the computations performed in response to the message. Part of the computations is sending a message to the ancestor to be processed there and another part is calling an auxiliary local procedure to perform the additional computations not performed by the ancestor. The point is that the descendant object is provided the possibility to send a privileged message to the ancestor object whose response is the evaluation of the ancestor's auxiliary local procedure.

6 ILLUSTRATIVE EXAMPLES

Two examples are given here, the first illustrates the use of our solution of multiple inheritance in addition to illustrating many of the features of the integrated SICStus Prolog system. Next an example is given showing our solution to the Self problem.

6.1 Inheritance and redundant computations

The following example is taken, with slight modification, from [Carnese 1984]. Using multiple inheritance, the following four classes are implemented:

The class **point** represents a movable point in one-dimensional space. An instance of *point* has the state *Point_Location* holding the current location of the point which is initialized, by default, to the value *0*. Each instance of *point* accepts the following message protocol:

- *location(Loc)* : bind *Loc* to the current location of the point.
- *move(NewLoc)* : change the current location of the point to *NewLoc*.
- *display(Device)* : print the string
  'Point at *Loc*' on the given output device where *Loc* is the current location of the point.

The following initialisation command is also provided:

- *init_at(X)* : bind the initial location of the point to the value of *X*.

Coded in the notations of the integrated SICStus Prolog system:

```
:- class(point, [location/1, move/1, display/1],
                [init_at/1]).

:- use_library(file_i_o, [writeln/2]).

%       load in the predicate 'writeln/2' from the library
%       module named 'file_i_o'

initialize([], [at(0)]).
initialize([init_at(X)], [at(X)]).

point(at(Loc)) :-
      location(Loc)   =>   point(at(Loc))
    ; move(NewL)      =>   point(at(NewL))
    ; display(S)      =>   writeln(S, ['Point at ', Loc]),
                           point(at(Loc)).

:- endclass(point).
```

The class **'history_point'** holds a list containing a record of all locations the *point* has had since its creation and accepts the following message protocol:

- *location(Loc)* : bind *Loc* to the current location of the point.
- *move(NewLoc)* : change the current location of the point to *NewLoc*. Update the history record.
- *display(Device)* : print the string:
  'Point at *Location*
  with location history: *HistoryRecord*' on the given output device.

The following initialization command is also provided:

- *init_at(X)* : bind the initial location of the point to the structure *at(X)*.

The class *history_point* is defined by inheriting the behaviour of the class *point*. The external interface of *point* is inherited but processing both messages *move/1* and *display/1* need to be redefined (overriding the ancestor class messages). To illustrate how an object may create and make use of other objects, the state of the location history record is represented as an instance of the class *queue* (defined above) onto which new locations are enqueued:

```
:- class(history_point, [location/1, move/1, display/1],
                        [init_at/1]).

:- inherits(point, all, all).

:- descends([print_history/1]).

% considering that the class definition may be inherited
% by other classes, the local message print_history/1 can
% be received only from descendants.

% initialize the record of location history by creating an
% instance of a queue with one element initially enqueued
% and with aQueueStream connected to the created
% instance of history_point.

initialize([], [QueueS]) :-
        make_instance(queue, QueueS, [with([0])]).
initialize([init_at(X)], [QueueS]) :-
        make_instance(queue, QueueS, [with([X])]).

history_point(QS) :-
      display(D)        =>  super(display(D)),
                            self(print_history(D)),
                            history_point(QS)
    ; move(L)           =>  super(move(L)),
                            QS=[enqueue(L)|QS1]),
                            history_point(QS1)
    ; print_history(D)  =>  QS = [display(H)|QS1),
                            print_history(D, H),
                            history_point(QS1).

print_history(D, History) :-
        writeln(D, ['with location history:', History]).

:- endclass(history_point).
```

Note that the computations performed in response to the message *display/1* have been factored so that the message *display/1* is sent to the ancestor and the local processing is performed by sending the privileged message *print_history/1* to self. Note that messages to self are given priority over other messages received by the object.

The class **'bounded_point'** represents also a *point* which has, in addition to a location, lower and upper bounds for that location. An instance of the class *bounded_point* accepts and processes the following messages:

- *location(Loc)* : bind *Loc* to the current location of the point.
- *move(NewLoc)* : change the current location of the point to *NewLoc* only if the value is in bounds.
- *display(Device)* : print the string
  'Point at *Location*
  with bounds: (min: *Min* **, max:** *Max* **)**' on the given device.
- *bounds(LB, UB)* : binds *LB* and UB to the current lower bound and the current upper bound for the point respectively.
- *set_bounds(NewLB, NewUB)* : changes the current bounds to *NewLB* and *NewUB* respectively.

The following initialization command is used:

- *set_bounds(InitLB, InitUB).*

```
:- class(bounded_point,
          [location/1,move/1,display/1, bounds/2,
                                        set_bounds/2],
          [set_bounds/2]).

:- inherits(point, all, []).
```

```
:- descends([print_bounds/1]).

:- use_library(file_i_o, [writeln/2]).

initialize([], [min(0), max(100)]) :-
        init_super(init_at(0)).
initialize([set_bounds(LB, UB)],[min(LB),max(UB)]) :-
        init_super(init_at(LB)).

% init_super/1 is a system-defined procedure used to
% ensure correct initialization of ancestor instances. It can
% use the ancestor's initialization command even those
% hidden from the clients of the descendant.

bounded_point(Min, Max) :-
      display(D)  =>  super(display(D)),
                      self(print_bounds(D)),
                      bounded_point(Min, Max)
   ; bounds(LB, UB)
                  =>  Min = min(LB),
                      Max = max(UB),
                      bounded_point(Min, Max)
   ; set_bounds(NewLB, NewUB)
                  =>  Min1 = min(NewLB),
                      Max1 = max(NewUB),
                      bounded_point(Min1, Max1)
   ; move(L)      =>  within_bounds(L, Min, Max),
                      super(move(L)),
                      bounded_point(Min, Max)
   ; print_bounds(D)
                  =>  print_bounds(D, Min, Max),
                      bounded_point( Min, Max).

print_bounds(D, min(LB), max(UB) :-
       writeln(D, ['with bounds: ( min: ', LB, ' ,
                                   max: ', UB, ' )']).

:- endclass(bounded_point).
```

The last class '**bh_point**' also represents a *point* which has a location, lower and upper bounds for that location and a location history record. An instance of the class *bh_point* accepts and processes the messages and initialization commands making a union of the external interfaces of both classes *history_point* and *bounded_point* in addition to the following message:

- *display(Device)* : print the string
  '**Point at** *Location*
  with bounds: (min: *Min* **, max:** *Max* **)**
  with location history: *HistoryRecord*' *on* the given output device.

The new class will be defined using multiple inheritance from both classes *bounded_point* and *history_point*. The messages *'location/1'*, *'move/1'* and *'display/1'* are duplicated. Because *'location/1'* has the same functionality in both ancestor classes, it can be sufficient to delegate it to only one of the ancestors. the message *'move/1'* is delegated to both ancestors. It is, however, necessary to resolve the conflict caused by the functionality of the message *'display/1'*. Sending the message *display/1* to both ancestors has the undesirable side effect that the output string will have the form:

Point at *Location*
with bounds: (min: *Min* **, max:** *Max* **)**
Point at *Location*
with location history: *History Record*

The redundancy of displaying the location of the point twice can be avoided by making use of privileged messages. The designer of the class may choose to send *display/1* to one of the ancestor objects and a privileged message to the other object to perform the local computations required to complete the task.

```
:- class(bh_point,
         [location/1, move/1, display/1, bounds/2,
                                        set_bounds/2],
         [init_at/1, set_bounds/2]).

:- inherits(bounded_point, all, all).
:- inherits(history_point, all, all).

initialize([],[]).

bh_point :-
     move(Location)    =>
          super(bounded_point,move(Location)),
          super(history_point,move(Location)),
          bh_point
   ; display(Device)    =>
          super(bounded_point,display(Device)),
          super(print_history(Device)),
          bh_point.

endclass(bh_point).
```

To create an instance of the class *bh_point*, a call of the following form may be entered:

```
?- make_instance(bh_point, SplitStream, []), ... .
```

An instance of *bh_point* primarily contains two objects of *point* which are replaced by a single one to which messages sent from the different *DelegateStreams* are merged into a single input stream:

```
?-  bh_point(InS, SelfS, HistS, BoundS),
    history_point(HistS,SelfS, PointSH, QueueS),
    queue(QueueS, SelfQueueS, InitialQueue),
    bounded_point(BoundS,SelfS,PointSB,LB,UB),
    merge(split(PointSH, PointSB), PointS),
    point(PointS, SelfS, InitialPoint).
```

6.2 Delegation to Self

After asking the question "What granularity of inheritance is allowed for an object hierarchy?" Bobrow [1984] points out that the explicit message delegation used in Concurrent Prolog [Shapiro and Takeuchi 1983] "doesn't allow fine grained specialization" and illustrates his statement by an example. This problem is also known as the "Self problem". The example given in [Bobrow 1984] is used here to illustrate how self communication works in our model between instances in a hierarchy and to show that fine grained specialization is supported in the model. The class *movable_object* holds information about the coordinate position (X,Y) of the object and has a message protocol according to the following declaration:

```
:- class(movable_object, [..., move/2, ... ], [...]).
```

The message *move/2* takes a new coordinate position *(NewX, NewY)* and is processed as follows:

```
movable_object(pos(Xspos,Yspos)) :-
        ...  ...
        ;   move(NewX, NewY)     =>
                    self(erase(Xspos, Yspos)),
                    self(draw(NewX, NewY)),
                    movable_object(pos(NewX, NewY)).
```

Upon receiving a message *move/2*, the two messages (*erase/2* and *draw/2*) are sent one after the other onto *SelfStream*. If no inheritance is involved, *SelfStream* is connected to the instance of *movable_object* itself. The definition of the class *movable_object* includes code to perform the appropriate response to both messages.

Assume that a new class *square* is defined to inherit the class *movable_object* and it is defined so that an instance of *square* delegates the message *move/2*, when received by it, to the ancestor instance of *movable_object*. The *erase/2* and *draw/2* messages are processed by *square* instance itself.

A message *move/2* received by an instance of *square* is delegated to the ancestor *movable_object* instance. The ancestor will respond by sending the message *erase/2* followed by *draw/2* onto its *SelfStream* which is connected to the descendant instance of *square* (see Section 5). Both messages *erase/2* and *draw/2* are now received by the instance of *square* to be processed there. Both messages are given priority over any other messages sent to *square* instance from client objects. Note that in Bobrow's example the *erase* and *draw* were nullary operations where in our solution *erase/2* and *draw/2* must carry the actual position of the point to be processed.

7 RELATED AND FURTHER WORK

In comparison to the approach presented here the languages Vulcan [Kahn et.al 1987], A'UM [Yoshida and Chikayama 1987] and Polka [Davison 1988] are based on similar concepts which stems from the approach presented in Concurrent Prolog [Shapiro and Takeuchi 1983]. None of these languages provide support for full encapsulation or the ability to program objects with non-deterministic behaviour. Non-determinism allows, e.g., generate-and-test programs to be written in a communicating object-oriented style. Non-determinism also allows sending a variable as a message to an object which may lead to non-deterministic evaluation of the methods defined by the class. The syntactic sugar provided here to make programs less verbose is much simpler than the syntactic constructs defined by the compared languages. Our syntactic sugar is greatly influenced by Prolog's Definite Clause Grammars (DCG) which abstract much of the code needed to implement logic grammars and allow calls and definitions of Prolog procedures within the grammar rules.

There is a work in progress extending the current system to provide mechanisms to support handling exceptions, errors and incomplete knowledge, and also to support the dynamic replacement of program modules (i.e, the ability to modify programs while the system is running). The work also include the development of a computational model and a methodology for program development in the integrated system and the provision of a high level debugging tool.

There is also work in progress to use the integrated system in the design and development of an industrial large software system which will demonstrate the viability of the integrated system and will provides real figures about the performance of the system. It is also interesting to investigate mechanisms and implementation strategies which provide a support to describe and handle explicit parallelism and concurrent activities.

ACKNOWLEDGEMENTS

I wish to thank Seif Haridi for helpful remarks on previous versions of the paper which helped in clarifying the ideas presented here. I also wish to thank Robert Virding, Mats Carlsson and Per Brand for stimulating discussions and the referees of the paper for valuable comments.

REFERENCES

[Brand and Haridi 1988] Prolog for Discrete Simulation, Research Report, SICS.

[Bobrow 1984] If Prolog is the Answer, What is the Question?, *Proc. FGCS'84*, ICOT.

[Carlsson 1987] Freeze, Indexing and Other Implementation Issues in the WAM, In *Proc.4th ICLP*, MIT Press.

[Carlsson and Widen 1988] *SCStus Prolog User's Manual*, Research Report R88007, SICS.

[Carnese 1984] Multiple Inheritance in Contemporary Programming Languages, LCS/TR-328, MIT.

[Chikayama 1984] Unique Features of ESP, *Proc. FGCS'84*, ICOT.

[Davison 1988] Polka: A Parlog Object Oriented Language, Technical Report, Dept. of Computing, Imperial College.

[Gullichsen 1985] BiggerTalk: Object-Oriented Prolog, Technical Report STP-125-85, MCC.

[Kahn, Tribble, Miller and Bobrow 1987] Vulcan: Logical Concurrent Objects, in Research directions in object-oriented programming, MIT Press.

[Kahn 1987] Objects with state in Prologs with Freeze, Technical Report, Xerox PARC.

[Quintus 1987] Quintus Prolog Reference Manual version 10, Quintus Computer Systems, Inc.

[Shapiro and Takeuchi 1983] Object Oriented Programming in Concurrent Prolog, *New Generation Computing*, vol. 1, no. 1.

[Shapiro and Safra 1986] Multiway Merge with Constant Delay in Concurrent Prolog, *New Generation Computing*, vol. 4, no. 2.

[Snyder 1987] Inheritance and the Development of Encapsulated Software Systems, in Research directions in object-oriented programming, MIT Press.

[Yoshida and Chikayama 1987] A'UM - Parallel Object-Oriented Language upon KL1 -, ICOT.

[Zaniolo 1984] Object-Oriented Programming in Prolog, *Proceedings of Symposium on Logic Programming*, IEEE.