SCOOP
Structured Concurrent Object Oriented Prolog[1]

Jean Vaucher, Guy Lapalme & Jacques Malenfant
INCOGNITO, Dépt. d'Informatique et R.O., Université de Montréal,
CP 6128, Station "A", Montréal, CANADA H3C 3J7

ABSTRACT

SCOOP is an experimental language implemented in Prolog that tries to combine the best of logic, object-oriented and concurrent programming in a structured, natural and efficient manner. SCOOP provides hierarchies of object classes. These objects behave as independent Prolog programs with private databases which can execute goals within other objects.

SCOOP also supports parallel processes, synchronised by the exchange of messages. For simulation, a sequencing set and primitives concerned with simulated time are provided. Thus, SCOOP has the ability to describe structured dynamic systems and to encode knowledge.

The important features of SCOOP are 1) its lexical block structure designed to promote and enforce modularity and to allow verification and optimisation via a compiler, 2) its combination of familiar programming cliches: the concepts of Simula67 for macro-structuring of entities and those of standard Prolog (unification & backtracking) for local behaviour, 3) its provision for parallel activity with a clear distinction between static objects and dynamic processes and 4) its discrete simulation capability.

---

[1] This paper was presented at ECOOP'88 in Oslo.

1. INTRODUCTION

Prolog is a relatively new programming language based on the notation of formal logic and the concepts of theorem proving[Cloc81, Colm83, Ster86]. It stresses a declarative style of programming where a program is written as a set of **facts** and **rules** pertinent to the problem at hand and execution is viewed as attempts to prove the validity of queries. Prolog has shown itself to be excellent for symbolic computation and has found many applications in areas such as natural language processing and expert systems.

Prolog is a deceptively simple language whose power derives from the systematic exploitation of two fundamental concepts: unification and backtracking. *Unification* is a pattern matching operation which is used to select applicable rules and facts and to effect parameter transmission. Backtracking means that, in trying to answer a query, the inference mechanism tries all possible combinations of facts and rules. In other words, Prolog automatically builds programs from available procedures at run-time. As a result, it is not uncommon to find that a dozen well-conceived lines of Prolog are equivalent to several pages of code in more traditional languages.

The logical elegance of Prolog and the compactness of its code makes it tempting to use for large-scale general-purpose programming. However, Prolog was not conceived with software engineering aspects in mind and there are no standard features to support hierarchical decomposition and modularity. Prolog is also weaker than traditional imperative languages in other aspects. The most glaring deficiencies lie in the following areas:

- modularity and protection,
- state changes (there is no assignment in Prolog) and
- expression of parallel activity.

These are exactly the areas where object-oriented (OO) concepts are most effective. Using the **object** methodology, systems are conceived in terms of **classes** of interacting **objects** of various types each with its own attributes, state and capabilities [Stef85]. Local procedures and state variables are usually hidden or protected so that each class may be designed and understood in isolation then assembled according to behaviour without concern for internal details. Moreover, OO languages, such as Simula [Dahl70, Dahl72], can be designed to exhibit static block-structure, so that many aspects of program behaviour can be deduced from the text. This helps in understanding, debugging and compiling.

Often, a **class** may be delared to be a specialisation or subclass of another, thereby **inheriting** all the attributes of the superclass. The inheritance mechanism facilitates incremental development and allows the creation of generic software packages whose object classes can be extended and customised. In some languages, multiple inheritance from several superclasses is allowed forming inheritance networks rather than hierarchies.

Object-oriented programming is also a good descriptive base for parallel computing [Yone87]. The notion of self-contained independent object is very close to that of parallel **process** or **actor**. Furthermore, the concept of communication by message passing between objects and local interpretation of messages maps well onto distributed environments.

In view of the complementary strengths of the logic and object-oriented paradigms, an integration would appear to be fruitful. Scoop tries to do just that: it combines what we think is the best of logic, concurrent and object-oriented programming in a natural and efficient manner. The Scoop system is implemented in Prolog but the use of a compiler and a meta-interpreter gives us great freedom to experiment in the design of the syntax and semantics of the language.

In Scoop, classes represent independent Prolog programs can operate with the full power of unification and backtracking. Inheritance between classes is also provided. Each object has its own private database: initially an exact copy of the clauses defined in its **class**. Some clauses are immutable and fixed for all objects; but others, like dynamic predicates in Prolog, can be asserted and retracted by the object. The fixed (static) clauses act as **methods** whereas the dynamic ones act as **state** variables. It is also possible to parametrise objects by inserting some dynamic clauses into the local database of the object at creation time.

An object can invoke goals within other objects. Initially, only one object, the **main process** object, is active. This object can create other objects and send them queries. To allow multiple activities to proceed in parallel, Scoop supports processes, synchronised by the exchange of messages. On a sequential machine, processes are implemented by time-sharing.

Much of our experience with object-oriented programming had been acquired through simulation with Simula, an OO language which could be extended to serve as a discrete-simulation language. As a test of the suitability of Scoop for general-purpose programming, we extended it for simulation in the same way with a sequencing set and scheduling primitives.

In the next section, we shall review related language proposals which combine some aspects of the object-oriented, concurrent and logic programming paradigms. Each is different either in its objective, its features or its base-language. Put together, the proposals cover just about all parts of Scoop. It is therefore important to underline the particular approach embodied in our proposal:

1) We take Prolog as the base language and seek to extend it in the most useful way. We do not attempt to compare Scoop to designs which go the other way and graft logic (or object, actor...) features to other base languages [Robi82, Lalo87, Yone87]

2) We believe that Scoop manages to integrate usefully more concepts than any other Prolog-based proposal.

3) Our perpective is that of *software-engineering*: we are interested in structuring Prolog so that large programs may be written, understood and assembled in modular fashion. Thus, our proposal will stress static inheritance structure to fix the meaning of names and we will not allow the user to define more flexible inheritance schemes or meta-interpreters. Similarly, we shall aim for macro-parallelism suitable for operating system processes rather than trying to exploit massively-parallel architectures

The salient features of the Scoop object/logic integration are the following:

- **block structured syntax:** designed to promote and enforce modularity and to allow verification and optimisation via a compiler.

- **familiar programming cliches**: using the concepts of Simula67 for macro-structure and those of standard Prolog for local object behaviour. Hopefully, Scoop programs should appear familiar to both object and logic programmers.

- **parallelism**: with a clear distinction between static entites (objects) and dynamic computing agents (processes)

- **discrete simulation capability**.

Scoop is not a finished language; it is an evolving experimental vehicle meant to show the feasability and desirability of the combined approach as well serve as a test-bed for new ideas. Some aspect of the language as therefore quite arbitrary and subject to evolution.

The remainder of the paper is structured as follows. First, we survey consider related language-integration proposals. Next, we describe object definition in Scoop along with remote calls and qualification of object references. After that, we consider, concurrent programming and message passing. Then, the simulation features of Scoop are described. Programming examples follow and the implementation is briefly discussed.

2. RELATED WORK

Object-logic proposals can be divided into two categories depending on wether or not they consider parallelism. In the first object-oriented language, Simula67, objects were introduced to simulate parallel dynamic entities and Simula objects operated as coroutines. Currently, over 20 years later, it is surprising that many object-oriented language proposals concentrate on modularity and neither support real or quasi-parallelism. We consider first the proposals which integrate logic and objects without considering parallelism, then those for concurrent programming.

Object-oriented logic programming

Some of the benefits of object-oriented programming, namely class-dependent methods and inheritance, can be achieved with minimal implementation by using a particular coding discipline for the methods and a SEND predicate which searches an explicit ISA hierarchy . This is the approach taken by Kahn (1981), Zaniolo (1984) and Stabler (1986). However, users must be aware of the internal representation of objects and the notation is verbose. Further study of the proposals reveals that there is no provision for changing object state and no consideration of concurrency.

Objlog [Chou87] and LOOKS [Mizo84] implement objects as part of more complex knowledge representation languages and concentrate on flexible inference schemes. LOOKS realizes a blend of objects and logic programming with class/meta-class/instances having methods defined as Horn clauses. Meta-classes here deal with control knowledge

associated with a class, allowing the user to define the inference procedure to be used. Objlog is specifically oriented toward complex knowledge representation problems as architectural databases. Neither system considers parallelism and there is no structured syntax.

ESP is an extension of KL0, a Prolog like machine language developed at ICOT [Chik84]. An object in ESP is an axiom set which responds to messages by trying to refute the submitted proposition using its axiom set. Inheritance through class hierarchy is provided and "slots", representing time dependent state variables, may be associated with each object. In Scoop, we follow standard Prolog and keep state information as dynamic clauses.

SPOOL [Fuku86] is a proposal which syntactically ressembles our own. SPOOL is built on top of IBM Prolog. It a uses block-structured syntax for class declaration and supports inheritance hierarchies. Full backtracking is allowed. Objects communicate via a "send" primitive which retains the full unification capability of Prolog to allow such things as anonymous recipients. SPOOL has no parallelism and, like ESP, it uses instance variables to implement state.

Proposals with parallelism

Parallelism in logic programming is also receiving considerable attention. Two families of approaches may be identified. First, much work has been focussed on the AND/OR model of parallelism, exemplified by Parlog [Clar86], Concurrent Prolog [Shap83, Shap86] and others [Kahn86] . These languages are designed for fast execution on massively-parallel architectures and concurrency occurs at a *micro-level* of the individual term. The ability to backtrack has been seriously limited in order promote speed. The notion of object with changing state is implemented in a side-effect free manner as perpetually recursive procedures. Although logically sound, this approach is unorthodox from the point of view of traditional programming. Object-oriented inheritance can be implemented in these languages, but there is no syntax to support the methodology and the notation remains opaque [Shap83, Kahn86]. At the present, these must be considered to be machine-oriented object-logic languages.

The second family is best described by the generic title: *communicating sequential Prolog processes.* Delta-Prolog [Pere84] illustrates this type of approach. It is based on

Monteiro's distributed logic and uses communication primitives similar to those of Communicating Sequential Processes. Scoop may be described as an object-oriented extension of the principles of this second family. Communicating Prolog Units [Mell86, Mell87] also are object-oriented extension of the Delta-Prolog family. However, they differ from Scoop by giving to the user more flexibility in the definition of relationship between objects (as in Actors). Scoop prefers static inheritance scheme to ease program comprehension and coding for most situation. Another example of parallel object-oriented logic language is Object-Prolog [Doma86]. In Object-Prolog, a program is divised into worlds defined as "modularised unit of knowledge" (Horn clauses). Arbitrary inheritance patterns may be coded into worlds again at the price of more programming efforts and run-time inefficiency.

Finally, one should mention T-Prolog [Futo86] which allows discrete simulation in Prolog. However, there is no attempt to provide object structure.

3. SCOOP CLASSES AND OBJECTS

Scoop modularizes Prolog programs by grouping related predicates in class definitions. A class is a template for the creation of objects of the same type. All objects of a particular class will have a common set of predicates particular to them. These predicates are separated in two sets. The first set contains static predicates which are fixed and common to all objects of the same class. These predicates can be viewed as methods. The second set contains dynamic predicates which may have initial clauses common to all objects of the same class but can also have new clauses passed as parameter at object creation and/or asserted or retracted during program execution. Dynamic predicate clauses function as individual databases private to each object that can model changes of state. To improve SCOOP's performance, dynamic predicate clauses are limited to facts whereas static predicate clauses can be either facts or rules. This has not been found to limit expressive power.

Class definition syntax

In Scoop, a class is a set of dynamic and static predicate definitions which can inherit definitions from other classes. The syntax of a class is shown below along with a typical example:

```
class <name>
      (<dynamic predicates dcl> ).
is_a  <super_class name> .
dynamic.
      <dynamic facts>

static.
      <static predicate clauses>
begin
      < body >
end.
```

a) Syntax of Class

```
class person (     name/1, child/2,
                   sex/1, +needs/1 ).
dynamic.
    name (unknown).
    sex (male).
    needs (love) .
static.
    son (S)        :- child (S, male).
    daughter (D)   :- child (D,female).
end.
```

b) Typical **class** definition

Figure 1 - Class Declaration

The class *name* is simply an atom which identifies the class. The **is_a** clause indicates inheritance relationships between classes in the program. In the example, *person* has no super_class.

Dynamic predicates function as parameters and state variables. They are declared in the *dynamicdeclarationpart* following the class name. Here, the programmer must give the mode, name and arity (number of arguments) of each dynamic predicates. In the example, four dynamic predicates are declared: name, child, sex, and needs. *Name* is a predicate with one argument, *child* has two arguments, etc...

Dynamic predicates can be of one of two "modes": either "add" or "replace". "Add" mode for a predicate is specified by preceding it with the symbol "+", otherwise default "replace" mode is assumed. The mode governs the way the parameters in the object creation will affect the local database of the object. In the example, *needs* is of mode "add". In the next section on object creation, we shall show how "mode" helps in the implementation of default attributes.

The *dynamic* part of the class definition is where the programmer gives the initial state of the objects that will eventually be created from this class. This initial state takes the form of facts which are instances of the predicates declared in the dynamic declaration list. A declared dynamic predicate need not have initial facts but every initial fact must refer to a declared dynamic predicate.

The static part lists invariant facts and rules common to all objects of the defined class. No explicit declaration of name and arity is required. The difference between dynamic and static predicates is that new dynamic facts may be passed as parameters when an object is created; they can also be asserted and/or retracted during program execution. Static predicates cannot be modified in any way during the program execution. Defining the same predicate both as static and dynamic is an error in Scoop.

The <body> defines initialisation code to be executed upon object creation. With the exception of the class name and the keywords **class** and **end**, all parts of the class definition are optional.

Object creation

Classes are merely definitions. It is the objects created from the classes that will own local databases containing dynamic clauses and have the behavior outlined by the definition. In Scoop, an object is created with the new/3 predicate:

new(*Object_ref*, *Class_name*, *Dynamic_clause_list*)

Class_name must be instantiated to the name of a defined class. The Scoop interpreter creates a new object of that class and generates a unique identifier for it. This identifier is unified with the *Object_ref* argument of **new**. The newly created object will be said to be of class *Class_name*. An object can obtain a reference to itself with the primitive **thisobject**(X).

At object creation, the object's internal database will be initialized with all the dynamic clauses defined in the dynamic part of the class. The object will also have access to all its class' static predicates. In this section, we only consider the simple case of a class without a *super_class*; inheritance will be covered later.

Dynamic_clause_list is a list of facts compatible with the dynamic predicates declared in the class. These facts will be added to the internal database of the object after the initialization. Here the mode of the predicates is taken into account. If a clause (or a set of clauses in the list) refers to a predicate with *replace* mode, all the initializing facts for this predicate will be replaced by the new fact (or set of facts). On the other hand, if it refers to a predicate with add mode, the new fact (or set of facts) will simply be added *after* the initializing facts. The two modes correspond to the familiar Prolog operations of *consulting* and *reconsulting* files.

To illustrate the difference between modes, consider the effect of the following statement creating a new person according to the class defined in Figure 1:

```
new( Fred, person,
        [name(fred), child(ann,female), child(joe,male), needs(money)] )
```

The local database created for Fred would contains the following facts:

```
name(fred).          % replacing "name (unknown)."
sex(male).           % default value
needs(love).         % initial fact
needs(money).        % added to the previous default clause.
child(ann,female).   % this shows that several clauses for the same predicate
child(joe,male).     %       can be inserted at once.
```

Scoop objects function like Prolog programs. Launching the proof of a goal G within an object O is termed a *remote call* and is expressed as "O:G". In the case of the person Fred, the query "Fred:son(S)" would succeed with S=joe. Similarly, the query "Fred:needs(N)" would succeed twice with N=love then N=money. "Fred:sex(red)" would fail.

Inheritance and scope of predicates

When a class C_1 is defined by an **is_a** clause to be a subclass of another C_2, objects of class C_1 have access, not only to the predicates of C_1 but also to others inherited from C_2. This inheritance can be affected by the redefinition of predicates in sub-classes and we must clarify the scope or visibility of predicates in Scoop objects. This discussion will also be pertinent to the next section which covers the scope of predicates in *remote calls* between objects.

In the *person* example, there was no inheritance. To illustrate the scoping rules, we shall use the following contrived example where class C "**is_a** B" which "**is_a** A" .

```
class a(p1/1).
    static.
            p2(a).
            p3(a).
    end.

class b(p1/1).
    is_a a.
    static.
            p4(b).
    end.

class c.
    is_a b.
    static.
            p2(c).
            p5(c).
    end.
```

First, we introduce the concept of *inheritancechain.* The inheritance chain of an object is defined as the ordered list of classes beginning by the class from which the object was created followed by all the classes inherited through the **is_a** relationship. In the example, if X is an object of class C, the inheritance chain of X is [C, B, A]. C is said to be the lowest class in the chain and, A, to be the highest. B and C are said to be below A . With these definitions, we can proceed with the scoping rules. You should remember that the scoping rules of Scoop are static in the sense that when you know the context of a call or predicate invocation, you always can determine which clauses will be used to try the reduction (and the class where they appear) by direct inspection of the source program.

Scoping rules:

1. A predicate is said to be *defined* in a class if appears either in the static part of that class or in its dynamic declaration part.

2. A predicate is visible in its defining class.

3. A predicate visible in a class C1 is also visible in any class C2 immediately below C1, unless redefined in C2. By redefinition, we mean that a predicate with the same name and arity is defined in the class. A predicate visible in a class but not defined in this class is said to be inherited.

Briefly, a goal occuring in a given class context refers to the first definition (or redefinition) encountered starting at the class of the context and going up the inheritance chain. An example can illustrate the rules. In our example, here are the predicates visible in each class context:

in A:					
from a:	**p1**	**p2**	**p3**		
in B:					
from b:	**p1**			**p4**	
from a:		**p2**	**p3**		
in C:					
from c:		**p2**			**p5**
from b:	**p1**			**p4**	
from a:			**p3**		

Clause access: local & remote calls

An object is a context or a black box that keeps track of internal information and provides services to computing agents. Access by an object of one of its own predicates is called a *local call*. Access by an object of a predicate in another object is called a *remote call* and, in the simplest case, it is expressed as "Object:Goal" (It is worth noting here that both static and dynamic predicates can be accessed with a remote call).

In Scoop, inheritance and sub_classing means that an object can be viewed at various levels and at each level, different clauses are visible. Different forms of the calls allow access to all levels.To understand the variants of accessing, it is useful to define a computation context as a couple (Object, Class) where Class is the defining class of the Object or one of its super_classes. The couple (Object, Class) will also be referred by "Object as Class". In Simula, this notion of context is called the *qualification* of a reference. The syntax of a call takes one of the following forms:

remote calls

```
X: g
X as class_name: g
```

local calls

```
self: g
self as class_name: g
super: g
g              % actually a local call
```

X: g, where X denotes an object, is the simplest remote call. It means calling the goal **g** in the context of the object **X** at its defining class, that is the one used as class_name in the *new* statement.

X as class_name: g means calling the goal **g** in the context of object **X** viewed as an object of class *class_name*. *Class_name* is usually a super_class of the actual class of **X**, but it could also be a sub_class. The predicate invoqued for **g** is the definition of **g** visible from the*class_name*. **Self: g** and **self as class_name: g** are the same as the previous forms except that the keyword "self" designate the actual object within which the call is made (in the same fashion as Smalltalk). The use of **self** is a convenience; it is possible to to the same thing by retrieving a self-reference with **thisobject(S)** and using **S** instead of **self** thereafter.

Super: g means calling **g** in the context of the actual object but one class higher than the class in which "super: g" occurs.

Finally, in the body of a class C, simple use of a goal identifier G (defined in C or its super_classes) is equivalent to "self as C: G".

Assuming the previous definitons of classes **a**, **b** and **c** and the object X created with new(X, c, []), the following shows how to access some of the defined predicates (i) within the context of **b** and (ii) from a context external to X:

predicate accessed	from the context of **b**	from a context outside X
p1 of **a**	super : p1(_) self as **a** : p1(_)	X as **a** : p1(_)
p1 of **b**	p1(_)	X : p1(_)
p2 of **a**	p2(_)	X as **a** : p2(_)
p2 of **c**	self : p2(_)	X : p2(_)

Asserting and retracting dynamic clauses

In Prolog there is no assignment statement to alter state, but clauses may be added or removed from the database. In Scoop, we use the same mechanism. The predicates asserta/1, assertz/1 and retract/1 are provided for this purpose and only dynamic clauses may be asserted and retracted during program execution.

In order to protect locality and protection for an object's state, Scoop does not allow asserta/1, assertz/1 or retract/1 as goals in a remote call. This restricts modification of an object's database to its own context. Naturally, if an object wants to provide an access to the dynamic management of its clauses from other contexts, it can simply have static predicates in its class definition allowing it. This means that no object can modify another's state without its knowledge or permission.

Management and access of dynamic clauses is more complex and onerous than for static clauses. In particular, dynamic clauses are replicated in each object whereas static clauses exists in only one copy. It is good Scoop practice to declare as dynamics only the predicates that will be asserted, retracted at run-time or passed as parameters at object creation.

4. CONCURRENT PROGRAMMING IN SCOOP

In the preceding sections, Scoop classes and objects have been introduced. However, classes and objects are only static entities. The active agents in Scoop are the processes. Initially, there is only one active process: the main program. However, it is possible to create others dynamically to execute independent sequences of goals. Scoop processes execute in parallel, time-sharing the interpreter; they synchronize and exchange information by message passing.

Process definition

Process definition in Scoop is a direct extension of class definition. Here, the <body> in the **begin** part (see figure 1) is considered to be a sequence of goals that a created process will attempt to demonstrate in parallel with its creator:

The initial process of a Scoop Program is defined by a "main class" where the "begin" part is mandatory and this "main" class must appear as the very last class definition in a Scoop program.

```
main (<dynamic declaration>).
dynamic.        …
static.  …
begin.
        <sequence of goals>
end.
```

Process creation and scheduling

Processes are created with the predicate new_process/3:

new_process(Process, Class_name, Dynamic_clauses_list).

As with the **new** primitive, this operation creates an object of class 'Class_name'. Facts in the 'Dynamic_clauses_list' act as parameters. Furthermore, an independent process, is created and launched to execute the sequence of goals in the context of the object. A unique process identifier is generated for that process (different from the object reference) and it is the process identifier which is returned in the 'Process' argument. Process identifiers can be used to effect communication via message passing (to be described later).

It is important to note that object references are distinct from process references. Objects refer to static entities whereas processes refer to dynamic executions of code. Initially, there is a strong association between a newly created process and the object serving as its initial context, but the process may move to execute in the context of other objects and several processes may be present in the same object at the same time. A process can obtain its own process reference with the primitive **thisprocess(P)** which unifies P to the unique reference of the current process. A reference to the object context where a process is currently executing is obtained via **thisobject(O)**.

Although we have chosen not to implement automatic mutual exclusion of processes within objects (as with monitors), it should be noted that the initial object of a process is private to the process and protected from any external tampering. This comes about because remote calls which could alter the object's state make use of object references and the object reference of a process is initially unknown. However, a process could change this by obtaining and broadcasting the reference of its object.

In the current implementation, Scoop processes are executed by time sharing. The interpreter repeatedly gives each process in a *ready_queue* a time slice until the queue is empty. Thus, processes in the *ready_queue* proceed in quasi-parallel fashion whereas

processes not in this queue are quiescent or passive. A newly created process is deemed active and placed at the tail end of the queue. Two scheduling primitives are provided to move processes in or out of the *ready_queue*. They are:

activate (Process)
and passivate (Process)

where Process must be a process reference.

Process Synchronisation

In order to perform coordinated computing, processes must be able to synchronize and exchange information. Scoop communication and synchronisation primitives are send/2 and wait/2:

SEND (<channel> , <msg>)
and **WAIT (<channel> , <msg>)**

The <channel> is used to select a receiver, whereas the <msg> is meant to carry the bulk of the information. For a successful message transfer to occur, both arguments in the **send** and the **wait** must match. The familiar appearance of these primitives masks some subtle points in the Scoop implementation. In particular, there is an important difference in the way that the "channel" and the "message" arguments are treated.

Send is non-blocking and it never fails. It creates a term of the form "msg(C,M)" with copies of **send**'s <*channel*> and <*msg*>parameters and places this term in a global message database. A process executing a **wait** is made to wait (if necessary) until a message with a unifiable <*channel*> parameter is available. When this occurs, the message is removed from the database and an attempt is made to unify the <*msg*> parameter. If this succeeds, the waiting process proceeds having extracted the message information via unification; if it fails, the **wait** operation is deemed to have failed and the process backtracks. Thus, a mismatch on <msg> fields can cause failure and backtracking but a mismatch on <channel> can only cause waiting. Note also 1) that a sent message can reactivate at most one waiting process and 2) that because messages are copies, the Concurrent Prolog technique of returning answers via un-instantiated variables in messages will not work.

An important aspect of the proposal is that that arbitrary terms are allowed for both arguments. In particular, although the <*channel*> could be a process reference and we are

not restricted to sending to either designated objects or designated processes. In the terms of SPOOL, Scoop can have have anonymous recipients. A few examples will show the flexibility of the mechanism. Below, the <channel> is a constant denoting the type of service required. There could be multiple servers.

Sending process:	**Receiving process:**
send(print, int(123)),...	...wait(print, int(X)),...

In the next example, a message is sent to a known Process and the built-in predicate, *gensym*, is used to generate a unique return address for the reply. This address is sent as part of the original message.

SENDER:	**RECEIVER:**
gensym(Ret_Id)	thisprocess (P),
send(Process , info(Input, Ret_Id)),	wait (P, info(X, Ret)),
	... < compute answer > ...
wait(Ret_Id, ans(Output)),...	send (Ret, ans(...)),...

For simulation, messages can also be used to implement the *seize* and *release* operations on resources:

```
seize(R)  :-    wait (R,1).
release(R) :-   send(R,1).
```

5. SIMULATION

Our model for object-oriented programming, Simula67, exhibited the power of its methodology by being a general-purpose language which could be extended easily to handle discrete-event simulation. After implementing a first version of Scoop called POOPS [Vauc86], we decided to extend it for simulation and created SIMPOOPS [Vauc87]. The extention was found to be trivial and Scoop retains the simulation features.

In discrete-event simulation, there is implicit sequencing between activities based on the concept of simulated time for events. It is ironic that the main difficulty we encountered in this extention, was limiting the inherent concurrency of Scoop processes to ensure that only one process was active at one time.

Essentially, we extended the original scheduler with a sequencing set (SQS): that is a queue of doublets < Process, scheduled event_time > ordered by increasing event_time. Now, when the ready_queue is empty, the interpreter looks into the sequencing set and transfers the processes scheduled at the next instant of simulated time into the ready_queue. It also updates its internal clock to the new time. Program execution ends when both the ready_queue and SQS are empty. Three new primitives were also added:

- **hold (DT) :**

  Causes the executing process to suspend itself for DT units of simulated time. This is implemented by removing the active process from the ready_queue and placing it in the sequencing set according to its scheduled reactivation time. The execution of **hold** prevents backtracking to previous goals.

- **time (T)** :

  Unifies T with the current simulated time.

- **terminate :**

  Cancels all current and future events and stops the simulation. This is useful when there is cyclic activity in a system and quiescence is never achieved.

Other types of synchronisation are easily implemented through the built-in SEND and WAIT primitives (i.e. SEIZE and RELEASE). In addition to these, other predicates such as **uniform(A,B,Ts)** were added to generate various random distributions. In all about 30 lines of code were required to implement all the simulation primitives.

6. EXAMPLES

The first example below first presents a MUTEX class which implements "semaphore" operations for mutual exclusion in a concurrent environment. Next, we show the implementation of a STACK data type. A local predicate, *pile(X)*, stores the stack state and the implementation uses **assert**/**retract** to modify state. To ensure correct operation with parallel processes, STACK is defined "as_a" mutex object and non-atomic predicate bodies are bracketed by calls to P and V to ensure mutual exclusion.

```
class mutex.
        static.
                p :- thisobject(O), wait(O, 1).
                v :- thisobject(O), send(O, 1).
        begin.
                v.      % initialisation
        end.

class stack.
        is_a    mutex.
        dynamic.
                pile([]).
        static.
                push (X) :-  p, retract( pile( S ) ),
                                assert ( pile( [X | S] ) ),   v.
                pop   (X) :- p, retract( pile( [X | S] ) ),
                                assert ( pile ( S ) ),          v.
                top   (X) :-    pile( [ X | S ] ).
                empty   :-      pile ( [] ).
        end.
```

Figure 2 - Stack Definition

Stacks can be created and used as follows:

```
new(St,stack,nil), St.push(1), St.pop(X), write(X),...
```

Figure 3 shows processes executing in parallel. **Scan** objects print out the leaves of binary trees they are given at creation time. The parameter *margin* serves to format the output. The main program creates two **scan** objects as processes and the output shows the parallel execution.

```
class scan (margin/1,tree/1) .
static.

    trav(t(L,R)):- trav(L), trav(R) .
    trav  ( X ) :-
          integer(X),
          margin(M), tab(M),
          writeln(X).
begin.
   tree(T), trav(T).
end.

main.
    begin.
      new_process(_,scan, [1 ,  t(t(1,t(3,5)),t(6,99))] ),
      new_process(_,scan, [10 , t(t(1,2),t(3,4))] ).
    end.
```

The Output:

```
1
      1
      2
3
5
      3
      4
6
99
*** THE END ***
CPU  : 0.77 sec
```

Figure 3. Parallel processes

The next example in Figure 4 shows interprocess communication using the **send** and **wait** primitives. The example is adapted from [Dahl72]. There are two producer processes similar to the processes of Figure 3. Each generates an ordered sequence of integer values terminated by the constant **eof** and sends them along a communication channel. The merge process merges values from both channel to keep the output ordered. As in [Shap83], the

merge operation is implemented as a tail recursive procedure with the cut (!) operator to prevent needless growth of the backtrack stack.

```
class prod (ch/1,tree/1) .
static.
        trav (t(L,R)):- trav(L).
        trav (t(L,R)):- trav(R) .
        trav ( X ) :-   X is_a number,
                        ch(C), send(C,X).

        go:- tree(T), trav(T).
        go:- ch(C), send(C,eof).
begin.
   go.
end.

class merge.
static.
        merge(eof,eof):-        !.
        merge(V1,eof):-         writeln(V1), wait(1,V1x),!, merge(V1x,eof).
        merge(eof,V2):-         writeln(V2), wait(2,V2x),!, merge(eof,V2x).
        merge(V1,V2) :-         V1<=V2, writeln(V1),
                        wait(1,V1x),!, merge(V1x,V2).
        merge(V1,V2) :-         V2<V1, writeln(V2),
                        wait(2,V2x),!, merge(V1,V2x).
begin.
        wait(1,V1), wait(2,V2), merge(V1,V2).
end.

main.
begin.
   new_process(_,prod, [ch(1),   t(t(3,7),33)] ),
   new_process(_,prod, [ch(2) ,  t(1,t(6,99))] ),
   new_process(_,merge,[]).
end.
```

The Output:

```
1
3
6
7
33
99
*** THE END ***
```

Figure 4. Synchronised Merge

```
class client (id/1).
```

```
static.
        seize(R)        :- wait(R,1).
        release(R)      :- send(R,1).

begin.
        uniform(0,10,Ta),   hold(Ta),
        id(N), Nx is N+1,   new_process (_, client, [id(Nx)] ),
                write(N),  writeln(" Waiting"),
        seize (res),
                write(N),  writeln(" Entering resource"),
                uniform(0,8,Ts),
                hold(Ts),
        release (res),
        write(N), writeln (" Leaving system").
end.

main.
begin.
        send(res,1),
        new_process( _, client,  [id(1)]  ),
        hold (20),
        writeln("Closing down system"),
        terminate.
end.
```

OUTPUT:

```
1 Waiting
1 Entering resource
2 Waiting
3 Waiting
1 Leaving system
2 Entering resource
4 Waiting
2 Leaving system
3 Entering resource
3 Leaving system
4 Entering resource
Closing down system

*** THE END ***
CPU  : 0.95 sec
EVALS: 3779
FAILS: 453
```

Figure 5. A single server queue simulation.

Figure 5 shows a typical single server queueing simulation and the output generated. Each arriving customer generates his successor and provides him with a unique identifying number. Inter-arrival and service times are random. Seize and release operations have been

implemented trivially via message passing. The main program controls the simulation. It starts the first customer and eventually shuts down the simulation. The output trace shows the interleaving of the activities of the various objects. The statistics at the end of the listing show that the execution of this program took about one second on a VAX8600.

```
CLASSES
        class(client)
        class(main)

STATIC PREDICATES
        static_predicate(client,seize)
        static_predicate(client,release)

DYNAMIC PREDICATES
        obj_pred(client,[id])
        obj_pred(main,[])

CLAUSES
        1: clauses(client,seize(_89),[wait(_89,1)],_90)
        2: clauses(client,release(_81),[send(_81,1)],_82)
        3: clauses(client, begin, [uniform(0,10,_70), hold(_70), id(_71),  _72 is  _71+1,
            new_process(_,client, [id(_72)]), write(_71), writeln( Waiting), seize(res),
            write(_71), writeln( Entering resource), uniform(0, 8,_73), hold(_73),
            release(res), write(_71), writeln( Leaving system)], _74)
        4: clauses(main, begin, [send(res,1), new_process(_,client, [id(1)]), hold(20),
            writeln(Closing down system), terminate], _45)
```

Figure 6. Simulation - Compiler Output

This program, which apes the Simula or GPSS style, looks familiar and is easy to understand. On the other hand, the Prolog clauses produced by the compiler to drive the interpreter are quite cumbersome. They are shown in Figure 6. Many object-oriented Prolog systems require users to program in this style.

Finally, in Figure 7 shows an extract from the largest Scoop program written so far: about 600 lines of Scoop which combined with another 500 lines of pure Prolog implement a small expert system shell with a multi-window graphic interface. The figure shows part of the definition of a dialogue window which is defined as a parallel object to allow multiple simultaneous interactions. Of interest is the "rect(....)" dynamic clause which specifies a default size and position for dialogue windows. This work by Augustin Paar, an MSc Student in our Department, showed that Scoop was a useful and practical programming

tool for large programs. By using a blend of Prolog and Scoop, acceptable interactive response could be achieved.

```
class obj_parallele (id/1).
is_a obj.
static.   . . .
end.

class dialogue(modal/1, title/1, shape/1, rect/4).
is_a    obj_parallele.
dynamic.
        rect(50,50, 150,350).
static.
        send(Showdialog) :- id(Id),  senddialog(Id, Showdialog).
        select :- id(Id),  d_select(Id).
        hide :-  id(Id),  d_hide(Id).
        ...etc...
end.
```

Figure 7. Extract from SCOOP Graphic Expert

7. IMPLEMENTATION

Scoop is a compiler/interpreter system implemented in Prolog. Different versions have been implemented with several brands of Prologs and run on Vax, Sun and MacIntosh computers. At present the program is about 450 lines of code divided evenly between interpreter, compiler and formatted trace output. More details on the language and its implementation are available in [Vauc86 and Vauc87].

One of the main problems in a concurrent/logic integration is the combination of backtracking and parallelism. The problem lies in the implementation of backtracking to undo the variable bindings (instantiations). Typically, a meta-interpreter for Prolog undoes bindings in the reverse order to which they were made. When one tries to simulate co-routines or parallel behaviour, this is no longer the case. Our solution implements both backtrack and parallelism by exploiting a simple fact. When a clause is **asserted** and added into the database, a new copy of each of its variables is created. To backtrack, it suffices to retrieve the copy from the database to restore the state to what it was. This method of *freezing* and *melting* variables is also described by Sterling and Shapiro [Ster86]. This

method of keeping track of state was forced on us by the use of C-Prolog which does not tail-recursion optimisation and forced us into a failure-driven interpreter loop.

Presently, our main concern is speed. The Scoop interpreter is about 60 times slower than the underlying Prolog interpreter, i.e. about 100 LIPS versus 6000 LIPS. Recent tests have shown that much of the overhead is due to the maintenance of backtrackable states. However, speed can be traded for granularity of parallelism. Any term which has no Scoop definition is assumed by the interpreter to be implemented in the underlying Prolog and passed on to the native Prolog interpreter. By judicious mixing of Scoop and Prolog, reasonable levels of performance, modularity and parallelism can all be achieved.

We have improved the compiler so that most method-lookup is done at compile time. Moreover, we are modifying the interpreter to take advantage of a new Prolog with tail-recursion and we plan to modify an existing Prolog engine to implement Scoop directly.

8. CONCLUSIONS

Scoop integrates concurrent, logic and object-oriented programming paradigms in a natural way. Thus, Scoop has the ability to describe structured dynamic systems and to encode knowledge. The important features of Scoop are:

1) provisions of both local logic programming and global object-oriented structure
2) lexical block structure designed to promote and enforce modularity and to allow verification and optimisation via a compiler,
3) use of familiar programming cliches: the concepts of Simula67 for OOP and those of standard Prolog (unification & backtracking) for local behaviour,
4) provision for parallel activity with a clear distinction between static entites (objects) and dynamic computing agents (processes)
5) acceptable performance and
6) discrete simulation capability.

At present we are using Scoop to implement a better distributed system with a remote graphic interface server. Scoop will also serve as a vehicle for language experimentation. In particular we will experiment with alternatives to multiple-inheritance, study various ways

to hide local information and enforce more protection. However, our main concern is speed and we plan to modify an existing Prolog engine to implement Scoop directly.

9. ACKNOWLEDGEMENTS

This research was supported in part by the Natural Science and Engineering Research Council of CANADA. The authors also wish to thank Charles Giguère, Director of the Computer Research Institute of Montréal (CRIM), who provided the computer facilities for the development of the early versions of Scoop.

10. REFERENCES

[Clar86] Clark K. and Gregory S. **Parlog: Parallel Programming in Logic**, *ACM Trans. on Programming Languages and Systems*, **8**, 1 (Jan. 1986), pp. 1-49.

[Chik84] Chikayama, T. **Unique Features of ESP**, *Proc. International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, pp. 292-298, (1984).

[Chou87] Chouraki, E. and Dugerdil, Ph. **The Inheritance Processes in Prolog: Multiple vertical with point of view and multiple selective with point of view**, in *GRTC technical paper #GRTC/187bis/Mars 1987* (CNRS Marseilles).

[Cloc81] Clocksin, W. & Mellish, C. **Programming in PROLOG**, Springer-Verlag, Berlin, (1981).

[Colm83] Colmerauer, A. , Kanoui H. et Van Caneghem M. **Prolog, bases théoriques et développements actuels**, *Techniques et Sciences Informatiques*, Vol. 2, N° 4, pp. 271-311, (avril 1983).

[Dahl70] Dahl, O-J., Myhrhaug, B. & Nygaard, K. (1970). **SIMULA Common Base Language**, Publication S-22, Norwegian Computing Center, Blindern, OSLO.

[Dahl72] Dahl, O-J., Dijkstra, E. & Hoare, C.A.R. (1972). **Structured Programming**, Academic Press, London.

[Doma86] Domán, A. **Object-PROLOG: Dynamic Object-Oriented Representation of Knowledge**, SzKl Comp. Research and Innovation Center, 1986, 14 p.

[Fuku86] Fukunaga, K. and Hirose, S. **An Experience with a Prolog-based Object-Oriented Language**, in *Proc. of Object-Oriented Prog. Sys., Lang.*

and Applic. '86 (OOPSLA), ACM Sigplan Notices **21**, 11 (Nov. 1986), pp. 224-231.

[Futo86] Futó, Y. and Gergely, T. **Logic Programming in Simulation**, in *Transactions of the Society for Computer Simulation* **3**, 3 (July 1986), pp. 195-216.

[Kahn82] Kahn, K., **Intermission - Actors in Prolog**, in *Logic Programming*, (Eds. Clark, K.L. & Tärnlund, S-A.), Academic Press, London, pp.213-228, (1982).

[Kahn86] Kahn, K., Tribble, E.D., Miller, M.S. and Bobrow, D.G. **Objects in Concurrent Logic Programming Languages**, in *Proc. of Object-Oriented Prog. Sys., Lang. and Applic. '86 (OOPSLA), ACM Sigplan Notices* **21**, 11 (Nov. 1986), pp. 242-257.

[Lalo87] Lalonde, W.R. **A Novel Rule-Based Facility for Smalltalk**, in *Proceedings of ECOOP'87,* Bigre+Globule **54** (Juin 1987), pp. 193-198.

[Mell86] Mello, P. and Natali, A. **Programs as Collections of Communicating Prolog Units**, in *Proc. of ESOP '86, Springer-Verlag Lecture Notes in Comp. Science* **213**, pp. 274- 288.

[Mell87] Mello, P. and Natali, A. **Objects as Communicating Prolog Units**, in *Proceedings of ECOOP'87,* Bigre+Globule **54** (Juin 1987), pp. 233-243.

[Mizo84] Mizoguchi, F., Ohwada, H. and Katayama, Y. **LOOKS: Knowledge Representation System for Designing Expert System in a Logic Programming Framework**, in *Proceedings of the International Conf. on Fifth Gen. Comp. Sys.*, 1984, pp. 606-612.

[Pere84] Pereira L.M. and Nasr R. **Delta-Prolog: a Distributed Logic Programming Language**, Proc. International Conf. on Fifth Generation Computer Systems, ICOT, Tokyo, pp, 283-291

[Robi82] Robinson, J.A. & Sibert, E.E., **LOGLISP: Motivation, Design and Implementation**, in *Logic Programming*, (Eds. Clark, K.L. & Tärnlund, S-A.), Academic Press, London, pp.299-313, (1982).

[Shap83] Shapiro E. & Takeuchi A., **Object oriented programming in concurrent Prolog**, *New Generation Computing* , Vol 1, pp. 25-48, (1983).

[Shap86] Shapiro E. **Concurrent Prolog: a Progress Report**, *Computer,* **19**, 8, pp. 44-54 (Aug. 1986)

[Stab86] Stabler, E.P. **Object-Oriented Programming in Prolog**, *AI Expert* , October 1986, pp. 46-57.

[Stef85] Stefik M. & Bobrow D., **Object-Oriented Programming: Themes and Variations**, *The AI Magazine*, pp. 40-62, (1985).

[Ster86] Sterling L. and Shapiro E. **The Art of Prolog**, MIT Press, 1986.

[Vauc86] Vaucher, J.G. & Lapalme, G., (1986). **POOPS: Object-Oriented Programming in Prolog**, Pub. Nº 565, Département d'informatique et de R.O., Université de Montréal.

[Vauc87] Vaucher, J.G. & Lapalme, G. **Process-oriented simulation in Prolog**, *SCS Multi-Conference on AI and Simulation*, San Diego, Jan. 1987, pp.41-46, (1987), also available as Pub. Nº 604, Département d'I.R.O., Univ. de Montréal.

[Yone87] Yonezawa, A. & Yokoro, M. (Editors) **Object-Oriented Concurrent Programming**, The MIT Press, Cambridge, Mass., (1987).

[Zani84] Zaniolo, C. **Object-Oriented Programming in PROLOG**, *Proc. International Symposium on Logic Programming*, Atlantic City, IEEE, pp. 265-270, (1984).