# *LO* and Behold! Concurrent Structured Processes

Jean-Marc Andreoli and Remo Pareschi
ECRC, Arabellastrasse 17
D-8000 Munich 81, West Germany,
tel (49)-(89) 926990
jeanmarc@ecrc.de remo@ecrc.de

Bureaus cannot live without a host,
being true parasitic organisms...

A cooperative on the other hand
can live without the state.
That is the road to follow.
The building up of independant units
to meet needs of the people
who participate in the functioning of the unit.

– William S. Burroughs, "The Naked Lunch"

## Abstract

*We introduce a novel concurrent logic programming language, which we call LO, based on an extension of Horn logic. This language enhances the process view of objects implementable in Horn-based concurrent logic programming languages with powerful capabilities for knowledge structuring, leading to a flexible form of variable-structure inheritance. The main novelty about LO is a new kind of OR-concurrency which is dual to the usual AND-concurrency and provides us with the notion of* structured process. *Such OR-concurrency can be nicely characterized with a sociological metaphor as modelling the internal distribution of tasks inside a complex organization; this complements the external cooperation among different entities accounted for by AND-concurrency.*

## 1 Introduction

Actor languages [1] have been introduced to provide linguistic support for open systems [13]. The metaphor "programs as societies" [16] felicitously characterizes object-oriented programming as is possible in such languages: objects (i.e. actors) form a community of interacting, cooperating individuals. The purpose of this paper is pushing this metaphor one step forward, in that we aim to deal, in the context of object-oriented programming, with communities functioning on a more sophisticated principle: indeed, communities where not just simple individuals, but also *organizations of individuals* cooperate. Nowadays societies, with the complex tasks they have to perform, are in fact most naturally modelled in this way. There are good reasons why this extended metaphor would lead to an improvement of the Actor model of computation.

First, let us observe that an organization is characterized by essentially two kinds of behaviours:

- *internal distribution:* the situation when a complex task is dealt with by partitioning it in several subtasks distributed to different parts of the internal structure of the organization.
- *external cooperation:* the situation when an organization is not adequate at all for performing a certain task, although it does need the result of performing it; in this case, it will ask the cooperation of another organization with the necessary know-how; but the overall handling of the request may end up as being a complex task in itself.

For example, a company working in mechanical engineering may commission a CAD simulation to a software company: the exact details of the request will be formulated by its production division, while its administrative division will take care of the payment upon delivery of the product. As for the soft-

ware company, one of its divisions will first provide a "high level" prototype of the requested product; later on, another division will work on a more efficient "low level" implementation.

This distinction about the two options available for an organization to handle different kinds of tasks gives us an interesting insight on the sharing of knowledge in object-oriented systems. For we can think of an object as of an organization obtained by *composing* together different subobjects, which can interact in the fulfillment of complex duties: we identify this situation with the internal distribution of tasks. On the other hand, objects may also *delegate* to other (completely separate) objects tasks which fall without their degree of behavioural specialization: this is the case corresponding to external cooperation.

In its usual formulation, the Actor model does not make a distinction between these two possibilities, or rather accounts for everything in terms of the second one. In fact, all the sharing of knowledge is in general obtained by creating delegates, which may in turn refer to other delegates. Therefore, it becomes all too easy proliferating "bureaucrat" objects, which waste computing energy for the only purpose of delivering requests for tasks someone else is going to do; this increases the overall entropy of the system, and burdens it with a heavy communication protocol (see [23] for some recent considerations on this problem). By contrast, if the sharing of knowledge comes also from composing objects into objects, then the communication necessary for the internal distribution of a complex task in a given object can be achieved in a purely static manner, simply by permitting the relevant subobjects to trigger the methods needed to solve the corresponding subtasks. In other words, the given object *inherits* the necessary knowledge, as opposed to *delegating* the task.

In the model of computation we are going to propose, objects can be viewed as organizations capable of solving tasks both in terms of internal distribution and of external cooperation. Our starting point is the family of concurrent logic programming languages [20], where actors can be elegantly implemented as *AND*-concurrent, stream-communicating processes [22]. A simple extension in the logic underlying these languages, given by the introduction of a novel form of concurrent disjunction (denoted by the symbol @), will provide us with the capabilities of *structuring* such processes to support an "organization-like" style of programming. We have shown elsewhere [4, 2, 3, 5] that this extension finds a rigorous proof-theoretic counterpart in Linear Logic, a logic recently introduced to provide a theoretical basis for the study of concurrency [8]. The programming language *LO* (for Linear Objects), which we have designed according to these principles, is therefore characterized by two kinds of concurrency:

- the *OR*-concurrency embodied in the disjunction @, to account for internal distribution of subtasks

- the *AND*-concurrency embodied in the conjunction &, to account for external cooperation

Thus, the sociological duality in the behaviour of organizations is directly reflected into a corresponding logical duality in the operational semantics of the language.

In virtue of this more expressive syntax, we can write programs burdened by lesser run-time overhead than in standard Actor systems: in fact, objects are identified with structured processes, and many complex tasks can thereby be automatically partitioned internally to single objects, instead of requiring the creation of new processes. On the other hand, such a "low entropy" model of computation is not bought at the price of institutional rigidity: we shall see that our cooperating organizations are amenable to being modified in the course of events via addition, deletion and change of role of elements in their structure.

This paper is organized as follows. In Section 2 we shall introduce the syntax and the operational semantics of *LO*. In Section 3 a sample application will demonstrate how the novel notion of *OR*-concurrency adds powerful knowledge structuring capabilities to the process view of objects which can be modelled with concurrent logic programming languages. In Section 4 we shall briefly describe the current state of implementation of the system. Finally, in Section 5 we shall compare our approach with related work.

# 2 Concurrent Structured Processes

## 2.1 The Process View of Objects

Concurrent logic programming languages based on Horn logic [20] provide an elegant implementation of objects (i.e. actors) as *proof processes* communicating with each other via shared variables [22]. Such processes may be perpetual, and can be suspended and then resumed upon interaction with the user; change of state is achieved side-effect free by performing an inference step which replaces a subgoal by 0, 1 or more subgoals. Thus, as opposed to sequential logic programming languages like Prolog, which are *transformational* systems, in the sense that given a certain input they will compute a certain output and then terminate, concurrent logic programming languages are *reactive* systems, which continuously interact with the environment [12]. Alternatively, they could be characterized as *open* systems, in the sense of [13].

Let us rapidly sketch the notion of process characterizing this class of languages. In this framework, an object state transition is identified with an inference step in the proof procedure. Admissible transitions are specified as Horn clauses of the form

**old_state** :- **new_state.**

**old_state** and **new_state** are atomic formulae (literals) encoding, respectively, the initial and the final state of the transition. Typically, the predicates of these formulae will be the same, and represents the class of the object. The arguments of the formulae are the slots and communication streams of the object. For example,

```
point(In,5,7,Out)
```

encodes a point in a 2-dimensional space with coordinates `5,7` and communication streams `In,Out` (input and output, respectively). A clause

```
point([proj_x|In],X,Y,Out) :-
    point(In,X,0,Out).
```

specifies a transition, triggered by the message `proj_x` on the input stream, which resets the second coordinate of the point (i.e. applies a projection on the $X$-axis). In the case of the object above, firing of this transition would change its state as follows:

```
point(In,5,0,Out)
```

Similarly, objects can be, respectively, created and terminated by clauses of the form

**old_state** :- **new_state** & **created_state.**
**final_state.**

The first clause specifies a transition with creation of a new process while the second clause specifies the condition for terminating of a process.

The main problem with this approach is knowledge sharing: state transitions specified for given objects cannot be used by objects corresponding to instances of more specific classes. For example, the clause above, processing the message `proj_x` for *2-dimensional* points, cannot be applied to a *coloured 2-dimensional* point like the one encoded in the following atom:

```
point(In,5,7,red,Out)
```

In fact, unification is not possible between this atom and the head of the clause. To cope with this problem one can simulate delegation [22] or even add it as a primitive to the language [15, 7], but here we have in mind a more static, albeit fully flexible, solution.

## 2.2 The Principle of Organizational Inheritance

As should be clear, objects in Horn-based concurrent languages do not really qualify for the title of "organizations": for, as much as they are capable of evolving through time as complex human conglomerates can do, they are barred from doing this in the modular fashion which is an unescapable feature of real-life social behaviour — that is, they miss what makes subgroups of a given community capable of independent mutation while still remaining part of a larger social structure. Seen from a linguistic viewpoint, objects suffer from the expressive narrowness of being packaged within single predicates, where one cannot talk of evolution of a subpart without mentioning the remaining idle parts.

The shift in perspective we introduce with *LO* is simple and radical: objects are encoded as multisets (i.e. unordered lists) of independent units. For example, an instance of coloured 2-dimensional point can be represented as follows:

```
point, In, x(5), y(7),
colour(red), out(Out)
```

Given the order in which this multiset has been displayed, the first literal is meant to identify the corresponding object as a point, the second one encodes the input stream, the third and the fourth one correspond to the two coordinates, the fifth one refers to the colour and the sixth one to the output stream. Transitions will be consequently specified in an extended syntax with respect to that of Horn-clause logic, so as to be capable of applying them to *subparts* of objects. Indeed, firing of transitions will be inspired by the following **Principle of Organizational Inheritance**:

> *An organization can solve a certain task if one of its suborganizations can solve it.*

Operationally, this is achieved by allowing transitions as formulae with multiple literals in their head, connected together via the novel connective @; each literal identifies an object component which is going to be involved in the change of state. Thus, the transition for projecting two-dimensional points on the first axis can now be specified as follows:

```
point @ [proj_x|In] @ y(Y) <-
    point @ In @ y(0).
```

This transition can be applied to any object corresponding to a supermultiset (modulo unification) of the multiset of literals in the head, like the coloured two-dimensional point above. The object changes its state by replacing the literals in the head with those in the body. In terms of the "objects-as-organizations" metaphor, this means that a coloured two-dimensional point can project on the $X$-axis, since its subpart corresponding to the multiset

```
point, In, y(Y)
```

can do it.

## 2.3 Adding Structure to Processes

Formally, the evolution of a set of processes can be represented as the building of a proof tree. Each node in a tree encodes a state of a process and branches encode state transitions. Each transition has a unique initial state but may have 0, 1 or more final states. In Horn logic, each node of a tree is labelled with a single goal (an atom or a conjunction of atoms) and the allowed transitions are described in terms of two inference figures:

- **Duplication**

$$\frac{G_1 \qquad G_2}{G_1 \& G_2}$$

- **Propagation**

$$\frac{G}{A}$$

  if $A$ :- $G$ is a clause in the program.

For instance, a clause $p$ :- $q \& r$ can be triggered by any process in state $p$ and the corresponding transition is encoded in the tree as

$$\frac{\dfrac{q \qquad r}{q \& r}}{p}$$

This subtree has the following bottom-up reading: the initial state of this transition is $p$ and the final states are $q$ and $r$.

Seen from this standpoint, the main idea behind *LO* consists of enriching the structure of the proof trees, thus achieving a notion of "structured process" capable of supporting an "organization-like" style of programming. Instead of dealing with nodes labelled by single (conjunctive or atomic) goals, we associate nodes with multisets of goals, called "contexts". The syntax of the formulae and the proof inference figures are consequently enriched to take full advantage of the structure of the contexts. We shall distinguish three kinds of non-atomic formulae: *views*, *goals* and *methods*. They are built from atomic formulae, using four main connectives: &, @, $\leftarrow$ and $\Leftarrow$. The first two connectives correspond, respectively, to concurrent forms of conjunction and disjunction. The other two correspond to two different, albeit strictly related, forms of implication. As shown in [4, 2, 3, 5], all these connectives can be reconstructed in terms of Linear Logic connectives [10, 9]; in fact, & is the "additive" conjunction of Linear Logic, @ is the "multiplicative" disjunction, $\leftarrow$ is linear implication and $\Leftarrow$ is linear implication combined with the modality "of course". We also use the logical constant $\top$ ("top" in Linear Logic).

Let $A, V, G$ and $M$ be syntactic variables ranging, respectively, over atoms, views, goals and methods. $V, G$ and $M$ are defined from $A$ as follows:

$$V = A \mid V_1 @ V_2$$

$$G = A \mid \top \mid G_1@G_2 \mid G_1\&G_2$$

$$M = V \leftarrow G \mid V \Leftarrow G$$

A *program* is a set of methods. A *context* is a multiset of ground goals. Let $\mathcal{P}$ and $\mathcal{C}$ be respectively a program and a context, and let $[\mathcal{P}]$ be the (possibly infinite) set of all ground instantiations of elements of $\mathcal{P}$. We follow the convention of using the comma "," to denote multiset union, and to refer to a singleton multiset in terms of its single element. The inference figures of *LO* are as follows:

**I - Termination**

$$\frac{}{\top, \mathcal{C}}$$

**II - Splitting**

$$\frac{G_1, G_2, \mathcal{C}}{G_1@G_2, \mathcal{C}}$$

**III - Duplication**

$$\frac{G_1, \mathcal{C} \qquad G_2, \mathcal{C}}{G_1\&G_2, \mathcal{C}}$$

**IV - Propagation** (partial match)

$$\frac{G, \mathcal{C}}{A_1, \ldots, A_n, \mathcal{C}}$$

if $A_1@\ldots@A_n \leftarrow G$ is an element of $[\mathcal{P}]$.

**V - Propagation** (total match)

$$\frac{G}{A_1, \ldots, A_n}$$

if $A_1@\ldots@A_n \Leftarrow G$ is an element of $[\mathcal{P}]$.

Thus, the method $p@q \leftarrow (r@s)\ \&\ t$ can be triggered from a process in state, say, $\{p, q, u\}$ and the corresponding transition is encoded in the proof tree as

$$\cfrac{\cfrac{\cfrac{r, s, u}{r@s, u} \qquad t, u}{(r@s)\&t, u}}{p, q, u}$$

The final states of this transition are $\{r, s, u\}$ and $\{t, u\}$.

For simplicity sake, the inference figures above have been defined for the basic case of ground goals. However, it is easy (although formally tedious) to generalize them to the non-ground case via unification. As a matter of fact, the use of unification for variable instantiation will be assumed in the examples discussed later on.

Let us interpret the proof rules above in terms of processes. **I** (termination) tells us that a process can successfully terminate if its associated context contains the distinguished constant $\top$. **II** (splitting) makes @ synonymous with multiset union: a formula $G_1@G_2$ can be replaced in a context by $G_1$ and $G_2$. **III** (duplication) handles conjunctive goals by creating two subprocesses, much in the same way as in Horn-based *AND*-concurrent languages: the difference here is that a conjunctive goal can occur in the context of other goals, which have to be duplicated for each subprocess — as we shall see, quite a handy feature to account for the creation of new objects by *cloning* previously existing ones. As for the two propagation rules: **IV**, based on partial match, embodies the principle of organizational inheritance introduced in the previous section, by allowing triggering of methods whose head offers a *partial view* of the object it applies to (that is, methods containing knowledge usable by suborganizations of the organization corresponding to the whole object); **V**, based on total match, maintains the possibility of triggering methods whose head contains a *complete view* of the object, just as in Horn-based concurrent languages. Another way of characterizing the difference is in terms of class inheritance: methods can be triggered via propagation by partial match if they are in any superclass of the object; instead, with propagation by total match, they must be in its most specific class.

## 2.4 *OR*- and *AND*-concurrency

We want to briefly characterize in which sense the connective @ has a disjunctive reading. As we have seen, this connective is responsible for building up contexts (when it occurs in a goal: rule **II**) and for analysing contexts (when it appears in the head of a method: rules **IV** and **V**). Observe indeed that the language of Horn logic corresponds to the sublanguage of *LO* where the connective @ never occurs, and the connectives $\leftarrow$ and $\Leftarrow$ can therefore be collapsed together. In other words, contexts in Horn logic always contain one single element. By permitting non-singleton contexts, @ introduces a form of *OR*-concurrency, characterized by the following features:

- the process identified by a non-singleton context may evolve via alternative applications of the propagation rule **IV** to any of its subparts, including the whole context itself (or just the whole context if proof rule **V** is involved); that is, all the subcontexts of a context determine possible alternatives for process propagation.

- those elements of the context which are not involved in a step of process propagation at a certain stage of the proof, remain part of the new context; therefore, they *may* be involved in another step of process propagation at a further stage of the proof. This is the essential point about *OR*-concurrency: elements in the context associated with a certain proof process can lie dormant for a while but may at any time reenter the computation.

*OR*-concurrency of this kind is a new thing with respect to all other forms of disjunctive reasoning previously available in logic programming languages, concurrent and not; for instance, it is quite different from the disjunction available in certain Prolog implementation, which makes a disjunctive goal succeed if one of the disjuncts *separately* succeeds. Instead, with @ we wrap together in one unique process several different subprocesses each of which may represent a possible option for the evolution of the overall process. Also, there is clearly no relationship with "*OR*-parallel" logic programming languages, which are characterized by parallel execution of alternative search processes and are not concerned with problems of concurrency and communication.

Finally, before going into a more extensive example, here is a simple case which demonstrates the radically different nature of the interaction among processes induced by the two connectives @ and & (responsible, respectively, for *OR*-concurrency (internal distribution) and *AND*-concurrency (external cooperation)). In the following program to reverse a list, we use the usual infix notation `[X|L]` to refer to a list with head `X` and tail `L`, and the notation `#t` to refer to the distinguished formula $\top$. With this program, the computation is performed by building a context with two elements, one corresponding to a two-place predicate holding an input list and a buffer list, and the other to a one-place predicate holding the reversed list.

```
reverse(L,K) <= rev(L,[]) @ result(K).

rev([X|L],M) <- rev(L,[X|M]).

rev([],M) @ result(M) <= #t.
```

Here, calling a goal like

```
?- reverse([a, b],K).
```

will involve propagating (by total match) on the first clause and then splitting, thus creating a context containing the atoms

```
rev([a, b],[]), result(K)
```

Propagation (by partial match) will be then triggered twice again by using the second clause, until the context is rewritten to

```
rev([],[b, a]), result(K)
```

At this point, the atom `result(K)`, which has been "dormant" so far, can reenter the computation: in fact, the whole context can trigger the third clause, with the effect of instantiating the free variable `K` to the reversed list `[b, a]`.

Consider now what happens when evaluating the goal

```
?- reverse([a, b],K1) &
   reverse([c, d],K2).
```

The conjunction & creates two completely separate `reverse` processes; as they do not share any variable, no interaction whatsoever is established between the two processes, and each of them binds `K1` and `K2` to, respectively, `[b, a]` and `[d, c]`, just as expected. Thus, & plays a fundamental role of "isolating curtain", to prevent unwanted interaction between communities which are meant to remain structurally separate, whether they establish or not communication.

# 3 A Sample Application

The sample application that we now describe manipulates graphical objects in a 2-dimensional space. The application involves three kinds of structured objects:

- an *input device*

- an *output device*
- *drawings*, which receive data from the user via the input device and send back other data to the user via the output device.

These objects are interconnected as *AND*-concurrent structured processes. In Fig. 1, we have represented the objects as labeled boxes while the communication flow is represented by arrows. The communication is achieved by sharing of streams, i.e. potentially infinite, partially instantiated lists. Actually, we use here *channels* [24], corresponding to a more structured notion of streams than the simple list-based one. Channels are defined as follows:

- `[]` is the empty channel.
- If `M` is a term and `C` is a channel, then `[M|C]` is a channel (list).
- If `C1` and `C2` are channels, then so is `{C1,C2}` (fork).

## 3.1 The Input/Output Devices

We first define the user interface objects, which have a very simple structure.

### 3.1.1 The Input Device

This object interfaces the external input provided by the user and the channels internal to the system, which the user wants to access. It associates each of such channels with a name, i.e. a user-provided label. The user can then select a channel by its name. In this way, user's messages are dispatched to different labeled channels. We represent the input device as a context of the form

```
dispatch, name(L1,C1),...,name(Ln,Cn)
```

Each `Ci` is a channel; `Li` is its associated name. User's commands on the external input are interpreted as follows.

- `Msg>L` means: write the message `Msg` on the channel named `L`.
- `dup(L1,L2)` means: fork the channel named `L1` and give the name `L2` to the second branch (the other one keeps the name `L1`).
- `close(L)` means: close the channel named `L`.

Thus, we have the following methods:

```
dispatch 'read_input(I) <- dispatch(I).

dispatch(Msg>L) @ name(L,[Msg|C]) <-
    dispatch @ name(L,C).

dispatch(dup(L1,L2)) @ name(L1,{C1,C2}) <-
    dispatch @ name(L1,C1) @ name(L2,C2).

dispatch(close(L)) @ name(L,[]) <-
    dispatch.
```

In the first method, we use the backquote symbol ' to refer to system primitives, like input/output instructions, arithmetics etc., called before executing the body of the method. This syntactic convention has been adopted in the current implementation of *LO*.

Notice also that the names given to the channels have no meaning from the point of view of the system: in no way they act as object identifiers — only the channels themselves are. Their only purpose is to provide an interface for the user.

### 3.1.2 The Output Device

This object draws line segments on a real display device. We represent it as a context of the form

```
display, C1,...,Cn
```

where each `Ci` is an input channel, receiving messages as display instructions (i.e. drawing line segments). Forking a channel amounts to connecting a new input channel to the output device. Thus, we have the following methods:

```
display @ [line(M1,M2)|C]
    'draw_line(M1,M2)
    <- display @ C.

display @ {C1,C2} <-
    display @ C1 @ C2.
```

## 3.2 The Drawings

Drawings are complex structures whose behaviour is modularly accounted for in terms of *OR*-concurrency.

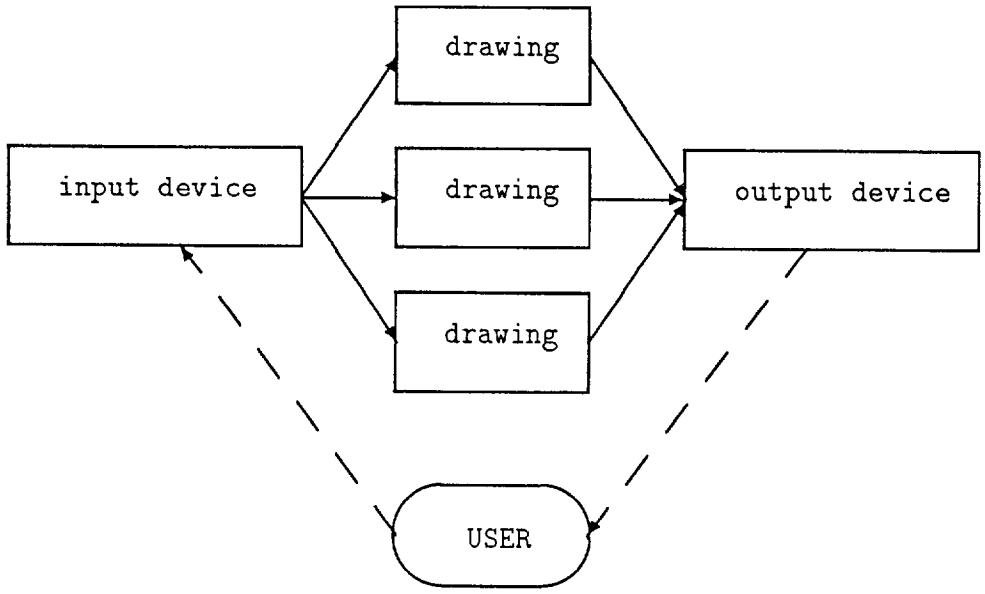


Figure 1: The objects and the communication flow

### 3.2.1 Fundamental Operations

A drawing contains, at least, the following basic components:

- a basic shape: square, circle etc.
- a scale, i.e. a 2x2 matrix specifying distortions (not only extensions and reductions) of the associated basic shape in each direction. Thus, many complex shapes can be obtained from basic ones by distorting them (all sorts of parallelograms from squares, and ellipsis from circles).
- a position, i.e. a point in the 2-dimensional space, which specifies a reference point for the basic shape (usually its center).

We encode these information in a context of the form

```
drawing, C, position(O), scale(P)
```

The variable C here represents the input channel.

The following operations are universal to all drawings, as they are only concerned with the position and scale slots:

- move(D) where D is a vector. This operation applies a translation of vector D to the drawing, thereby modifying its current position.
- distort(D) where D is a matrix. This operation applies an affin transformation of matrix D to the drawing, thereby modifying its current scale.

Thus, we have the following methods

```
drawing @ [move(D)|C] @ position(O)
    'O1 is O+D <-
    drawing @ C @ position(O1).

drawing @ [distort(D)|C] @ scale(P)
    'P1 is D*P <-
    drawing @ C @ scale(P1)
```

### 3.2.2 Graphical Operations

We now define the graphical operations themselves. They depend on the basic shape but also on the current position and scale.

For each basic shape, there is a method print capable of writing, on the *input* channel, instructions for a "pen" describing the specific shape. There are two types of pen commands:

- jump(X): which causes the pen to jump, without writing, to position X.

- `step(D)`: which causes the pen to move, while writing, to the position obtained by translating the current position by vector `D`.

In these commands, `X` and `D` are relative to the basic shape and do not pertain to absolute positions. For instance, for graphical objects which are squares — that is, encoded as contexts containing `square` among their elements — pen commands are generated by triggering the following method:

```
square @ [print|C] <-
    square @ [jump(m(1,0)),
    step(m(-1,1)),step(m(-1,-1)),
    step(m(1,-1)), step(m(1,1))|C].
```

Such pen commands are written in the input channel itself, so as to be executed immediately, before any other messages: further modifications of the drawing must take place after their execution. Fig. 2 illustrates the pen movements in the case of a square. The pen commands must then be converted into actual display commands, taking into account the current position and scale of the drawing, and be sent to the display device. The following two components perform that task:

```
pen(Q), out(U)
```

`Q` is the current absolute position of the pen and `U` is the output stream (towards the display device). We have then the following methods:

```
drawing @ [jump(X)|C] @ pen(_) @
position(O) @ scale(P) 'Q is O+P*X <-
    drawing @ C @ pen(Q) @
    position(O) @ scale(P).

drawing @ [step(D)|C] @ pen(Q) @
scale(P) @ out([line(Q,Q1)|U])
'Q1 is Q+P*D <-
    drawing @ C @ pen(Q1) @
    scale(P) @ out(U).
```

### 3.2.3 Prototypes and Clones

We now consider the case where the input channel of a drawing is a fork. We interpret it as a message to *clone* the drawing object which receives it, by creating an object with the same structure but different input and output channels. Thus, we add the following method:

```
drawing @ {C1,C2} @ out({U1,U2}) <-
    drawing @ C1 @ out(U1) &
    drawing @ C2 @ out(U2).
```

The capability of duplicating contexts because of proof rule **III** (duplication) is here crucially exploited to implement the cloning operation.

Once the clones have been created, they can be modified independently. For this, we already have the basic operations `move` and `distort` described in Section 3.2.1. However, we assume that initially, all drawings have a default basic shape, `noshape`, with no special `print` method; therefore we also need methods for changing the basic shape of such "invisible" entities. Here is an example, where a drawing is made square.

```
drawing @ noshape @ [make_square|C] <-
                drawing @ square @ C.
```

Once a basic shape (other than `noshape`) has been defined, it cannot be changed, nor other basic shapes can be assigned to the object; this is achieved by assuming that all shape-assigning methods only work for objects with the default shape `noshape`. In this way, we avoid the problem of dealing with such disquieting entities as "squared circles".

## 3.3 The System at Work

### 3.3.1 Initialization

Initially, three processes are created and connected:

- the input device
- the output device
- a shapeless prototype drawing, clonable as many times as needed

The input channel of the prototype is associated with some name, for example `proto`, to make it accessible to the user. The initial goal, which initializes the three processes, is therefore

```
dispatch @ name(proto,C) &
drawing @ noshape @ C @
position(m(0,0)) @ scale(m(1,0,0,1)) @
pen(_) @ out(U) &
display @ U.
```

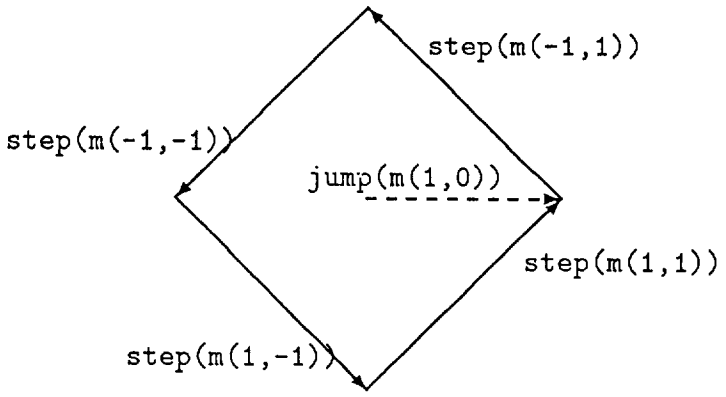


Figure 2: The pen movements for a square

Here is an example of user input data:

```
dup(proto,my_square),
move(m(200,300))>my_square,
distort(m(0,200,-100,0))>my_square,
make_square>my_square,
dup(my_square,an_other_square),
move(m(500,300))>an_other_square,
move(m(700,100))>my_square,
print>my_square, print>an_other_square,
....
```

The first line creates a clone of the prototype `proto` and names it `my_square`. The second and third line modify the characteristics of `my_square`. The fourth line defines it as a square. Given the distortion matrix, it will appear in fact as a rectangle. The fifth line creates a clone of `my_square` and names it `an_other_square`. The sixth and seventh line modify both `my_square` and `an_other_square`, which have now separate lives. The last line prints on the display device the two objects that have been created.

### 3.3.2 Execution

To obtain a proper behaviour of the system, we need to introduce some synchronization mechanism among the *AND*-concurrent processes. We rely here on the usual technique of suspending variables [20, 11]. We could introduce explicit "read-only" marks or mode declarations but we prefer the following convention: the application of a method is postponed if its head matches a subpart of the context which contains a free variable at the top level (typically the case of input channels). The method is actually triggered only when these variables become instantiated (if the instantiation is still compatible with the head of the method). When all candidate methods on a context are postponed, then the proof process for this context is suspended. When all processes have been suspended, the user can resume the computation by feeding the global input, thus causing some suspending variables to become instantiated (via the input device).

### 3.3.3 Termination

We still have to define methods to terminate the life of objects. Here, closing of the global input should make all the living processes terminate. This can be achieved via the following methods:

```
/* Input device */
dispatch(end_of_input) @ name(_,[])
    <- dispatch(end_of_input).

dispatch(end_of_input) <= #t.

/* Drawings */
drawing @ [] @ out([]) <- #t.

/* Output device */
display @ [] <- display.

display <= #t.
```

Notice here the use of the total match implication. It forces, for example, the input device to close all its named streams before terminating. Similarly, the output device terminates only when all its input streams are closed.

# 4 Implementation

An interpreter for executing *LO* programs has been implemented using the sequential logic programming environment SEPIA [17]. The coroutining facilities of SEPIA have been exploited to account for the specific control strategy described in Section 3.3.2.

Obviously, the interpreted approach is quite inefficient. Indeed, the propagation proof rules **IV-V** involve trying to match *each* head of a method with *each* subpart of the context, although, in general, very few matches will succeed. A smarter execution strategy should be capable of foreseeing many cases of failure, thus avoiding them at execution time. For instance, in the sample application of the previous section, it can be shown that none of the methods for the display device (characterised by the atom `display` in their head) will ever apply to a context representing a drawing (i.e. containing the atom `drawing`). This kind of analysis can be performed during a "compilation" phase and may improve significantly the efficiency.

A pre-processor based on this principle has been implemented. Using partial evaluation techniques, it computes a "type" for each context that may appear during the execution. Types are then used for two main purposes.

- The preprocessor associates to each type the methods whose head is compatible with that type. At run time, only those methods are tried on contexts of that type. This technique improves time efficiency.
- The preprocessor also tries to optimize the internal representation of the contexts of a given type, especially when it appears that such contexts have a fixed structure (typically, a specific number of slots of a class of objects). This technique improves space efficiency.

Notice that the interpreter corresponds to the degenerate case where all contexts have the same type. For space reasons, we cannot describe here the type inference procedure used in the preprocessor. It yields encouraging results with the simple applications that we have tested.

We are also exploring abstract computational models which could lead to efficient low-level implementation of *LO*. From this point of view, it seems particularly interesting to explore connections with the rewriting model recently proposed by Meseguer [18] as a unifying framework for concurrency. Finally, we are investigating whether system called (introduced with the backquote notation in *LO*) can be implemented as constraint as in Saraswat's concurrent constraint logic programming languages [19].

# 5 Discussion

Our starting point has been Horn-based concurrent logic programming languages. These languages identify objects with the proof processes of atomic goals; under this view, arguments in atomic goals are either stream or state parameters, where streams allow interobject communication, while states hold the value of the slots of the object in its current state. While this approach is very elegant and powerful from the point of view of message passing and dynamic modification of objects, it has strong drawbacks from the point of view of knowledge structuring, since, to change the value of a few slots, a method must explicitly access all the components in the object, even those which are going to be left unchanged in the new state of the object. This is a very undesirable situation — as it is easy to find applications with objects involving large numbers of slots — which can be partially obviated via the use of *delegate* processes as in [22]: that is, explicit structures are created only for objects with a manageable number of slots, and more complex objects are created by delegating to the simpler ones via stream communication. But, as distinguished from Actor languages (like Act1 [16]), where it is a built-in mechanism, here delegation has to be simulated by the programmer; higher-level languages built on top of Horn-based ones like Vulcan [15] and Polka [7] avoid this problem but have to introduce mechanisms like class declarations, which fix the structure of objects. (Actually, Vulcan has also class declarations which support

classical fixed-structure inheritance.) A more recent solution, known as *logic programming with implicit variables* [14, 21] is closer to the usual syntax of logic programs but still requires declaring in advance the slots of an object. The use of multiple head clauses has already been proposed in [6, 19]. These systems rely on an "extended" resolution mechanism, which can be reconstructed in *LO* using only the @ connective. Hence, in these systems there is no notion of multiple contexts (corresponding to different objects) as obtained in *LO* with the & connective (duplication rule **III**): all the objects must share the same, unique, resolvent (using explicit object identifiers to avoid name conflicts); when huge numbers of objects are involved, the resolvent may become untractable.

## Acknowledgements

We are grateful to Alexander Herold for helpful discussions on the subject of the paper. Thanks are also due to Henry Lieberman and the anonymous OOPSLA referees for comments which improved the final version of the paper, and to Gerard Comyn for his encouragement and support.

## References

[1] G. Agha and C. Hewitt. Actors: a conceptual foundation for concurrent object-oriented programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[2] J.M. Andreoli. *Proposition pour une Synthèse des Paradigmes de la Programmation Logique et de la Programmation par Objets.* PhD thesis, Université Paris VI, Paris, France, 1990.

[3] J.M. Andreoli and R. Pareschi. Formulae as active representation of data. In *Actes du 9eme Séminaire sur la Programmation en Logique*, Trégastel, France, 1990.

[4] J.M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In *Proc. of the 7th International Conference on Logic Programming*, Jerusalem, Israel, 1990.

[5] J.M. Andreoli and R. Pareschi. Logic programming with linear logic. In *Extensions of Logic Programming*, Lecture Notes in Artificial Intelligence. Springer Verlag, 1990. To appear.

[6] J.S. Conery. Logical objects. In *Proc. of the 5th International Conference on Logic Programming*, Seattle, USA, 1988.

[7] A. Davison. POLKA: a parlog object oriented language. Technical report, DOC, Imperial College, London, UK, 1988.

[8] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.

[9] J.Y. Girard. Quantifiers in linear logic. In *Atti del Congresso Temi e Prospettive della Logica e della Filosofia della Scienza Contemporanee*, Cesena, Italy, 1987.

[10] J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[11] S. Gregory. *Parallel Logic Programming in Parlog.* Addison-Wesley, 1987.

[12] D. Harel and A. Pnueli. On the developpement of reactive systems. In K.R. Apt, editor, *Logic and Models of Concurrent Systems.* Springer Verlag, 1985.

[13] C. Hewitt. The challenge of open systems. *Byte Magazin*, 1985.

[14] K. Kahn, W. Silverman, and E. Shapiro. Logic programs with implicit variables. unpublished, 1988.

[15] K. Kahn, E.D. Tribble, M.S. Miller, and D.G. Bobrow. VULCAN: logical concurrent objects. In E. Shapiro, editor, *Concurrent Prolog.* MIT Press, 1986.

[16] H. Lieberman. Concurrent object oriented programming in ACT1. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming.* MIT Press, 1987.

[17] M. Meier, P. Dufresne, and D.H. de Villeneuve. SEPIA. Technical report, ECRC, München, W. Germany, 1988.

[18] J. Meseguer. Rewriting as a unified model of concurrency. Technical report, SRI International, Menlo Park, USA, 1990.

[19] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Pittsburg, USA, 1989.

[20] E. Shapiro. A subset of concurrent prolog and its interpreter. Technical report, Institute for New Generation Computer Technology, Tokyo, Japan, 1983.

[21] E. Shapiro. The family of concurrent logic programming languages. Technical report, The Weizmann Institute of Science, Rehovot, Israel, 1989.

[22] E. Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog. *New Generation Computing*, 1(1), 1983.

[23] C. Tomlinson, M. Scheevel, and Won Kim. Sharing and organisation protocols in object-oriented systems. *Journal of Object Oriented Programming*, 2(4), 1989.

[24] E.D. Tribble, M.S. Miller, K. Khan, D.G. Bobrow, and C. Abbot. Channels: a generalisation of streams. In E. Shapiro, editor, *Concurrent Prolog.* MIT Press, 1986.