Communication as Fair Distribution of Knowledge

Jean-Marc Andreoli and **Remo Pareschi**

ECRC, Arabellastrasse 17
D-8000 Munich 81, Germany
{jeanmarc,remo}@ecrc.de

Abstract

We introduce an abstract form of interobject communication for object-oriented concurrent programming based on the proof theory of Linear Logic, a logic introduced to provide a theoretical basis for the study of concurrency. Such a form of communication, which we call forum-based communication, can be seen as a refinement of blackboard-based communication in terms of a more local notion of resource consumption. Forum-based communication is introduced as part of a new computational model for the object-oriented concurrent programming language *LO*, presented at last year OOPSLA/ECOOP (1990), which exploits the proof-theory of Linear Logic also to achieve a powerful form of knowledge-sharing.

1 Introduction

The programming language *LO* (for Linear Objects) [6, 4, 3] has been designed to supply a logical framework for object-oriented concurrent programming, with the purpose of rigorously accounting for its various aspects (concurrent communication, knowledge sharing, object creation, object termination etc.) in terms of the proof-theoretic behavior of logical connectives, thus making very simple and completely abstract the operational semantics of the system; its formal background is given by Linear Logic [11], a logic introduced by Jean-Yves Girard to provide a theoretical basis for the study of concurrency. *LO*'s view of active, concurrent objects as structured entities capable of sharing knowledge has on the other hand been illustrated in [4] in terms of a sociological metaphor: objects can be thought of as complex organizations, which inherit the problem solving capabilities of their suborganizations. Procedural knowledge is correspondingly encoded by specifying state transitions (*methods*) of the form

$$C_1 \parr \ldots \parr C_n \multimapinv Body.$$

where each C_i is an atomic logical formula. The logical (proof-theoretic) interpretation of methods hinges on the identification of objects, and of systems of objects, with proof trees and goes as follows: if, in constructing a proof tree in the Linear Logic calculus of sequents, we have reached an open node containing, among others, components $C_1 \; \ldots \; C_n$, then expand this node into another node obtained by replacing $C_1 \; \ldots \; C_n$ with the contents specified in *Body*, and leaving the remaining components untouched; this may lead to the creation of other nodes, or to a stop in the evolution of that branch of the proof, depending on the logical nature of the information provided by *Body*. The sociological interpretation goes instead as follows: if an organization contains in its current state a suborganization whose member elements are $C_1 \ldots C_n$ then let the given suborganization perform the task specified in *Body*, thus changing its own state and, consequently, that of the whole organization. According to both the logical and the sociological interpretation, objects evolve by explicitly performing actions: elements in the current state of affair may irreversibly disappear, to be replaced by new ones. On the sociological side this, and nothing else, could have been expected; but, on the logical side, such a capability to deductively deal with change in open worlds of independently coexisting entities specifically exploits the fact that Linear Logic accounts not just for truth, but also for the complementary notion of action, which had been instead neglected in more traditional logical developments. Thus, logic and sociology agree to each other, the first one giving us a rigorously defined abstract operational semantics for the language, and the other an anthropomorphic view of it which fits well within the tradition of object-oriented programming.

LO's organizational approach to knowledge sharing can be viewed as a form of *intraobject* communication, with objects acting as structured entities: for the capability of a subobject to handle a certain task is transmitted to the entire object in terms of the operational semantics of method triggering. We have argued that this form of inheritance does not suffer from the computational drawbacks of dynamic approaches to knowledge sharing like *delegation* [18], since it avoids the proliferation of delegate objects, which act as "bureaucrats" whose only purpose is delivering requests for tasks someone else is going to do; on the other hand, none of the malleability of delegation is lost, as *LO*'s objects have a completely flexible structure, where new types of components can be added at run time, while other ones may altogether disappear; nor does this entail populating our universe with anything more but simple "individuals" and their aggregations — abstract in its operational semantics, *LO* is quite concrete in its ontology: no notion of class is introduced, and the object-subobject relationship suffices to achieve inheritance (this can be contrasted with the class-superclass relationship characterizing class-based languages)[1].

However, another equally crucial and primary form of communication characterizes object-oriented concurrent programming: *interobject* communication, where separate entities exchange information, whether they be structured or not. This is the basic ingredient for object-based concurrency, particularly in languages of the Actor family [2], of which *LO* is an offspring on the side of its logic programming branch [27]; communication of such kind reduces procedure calls to exchanges of messages between objects, thus leading to a completely decentralized and truly concurrent model of computation. The computational model for *LO* presented in [6, 4, 3] has dealt with interobject communication simply by importing the technique of "shared logical variables" directly from concurrent logic programming languages based on Horn logic, like Concurrent Prolog [26], Parlog [12] and GHC [29]. However, such a solution is to be considered as temporary and not completely satisfactory mainly for the following (strictly related) reasons:

(*i*) It is committed to a specific implementation choice (the use of unification of logical variables in order to achieve communication) and thus defies *LO*'s effort towards a completely abstract operational semantics, which can be supported by different kinds of implementations and architectures.

(*ii*) It burdens the programmer with the task of dealing her/himself with problems related to stream-based communication, like stream-merging in many-to-one communication.

(*iii*) Although intraobject and interobject communication appear as conceptually dual, they are not so at the operational level: the intraobject case is accounted for in terms of proof construction, while unification handles the interobject case. The situation would become quite

[1] Obviously, we view classes and aggregations of individuals as quite different entities: the former are abstractions on individuals, the other are simply obtained by putting together simpler individuals to obtain more complex (composite) ones.

more pleasant by making proof construction capable of handling both cases; it would also be quite more in the spirit of Linear Logic, which is based on a system of dualities of logical operators.

In this paper, we provide a computational model for *LO* which refines the one described in the previous papers precisely by making proof construction responsible both for intraobject and interobject communication. In such a new model, proof construction is going to be seen as a *bidirectional* process, where, by starting from a *partially* defined initial node, we both go ahead in building the branches of the proof-tree and in further specifying its initial node. Thus, perhaps not surprisingly, dual communication concepts can both be accounted for via a fully symmetric approach to proof construction. This is obtained by permitting partially specified nodes to be instantiated upon the triggering of methods, which are now written as

$$C_1 \parr \ldots \parr C_i \parr \hat{} C_{i+1} \parr \ldots \parr \hat{} C_n \multimapinv \mathit{Body}.$$

Here, the components $\hat{}C_{i+1}, \ldots, \hat{}C_n$ (if any) are added to the unspecified part of the node once the method is triggered, while the components $C_1, \ldots, C_i$ must be found already there. But the unspecified part of any node gets percolated from the unspecified part of the initial node, and is therefore shared by all objects; consequently, whenever the triggering of a method by an object makes new components to be added to the unspecified part of the node, then these components are propagated back to the root of the proof tree and can in turn be used by other objects. To stick to the organizational metaphor, we can think of the unspecified part of the initial node as a kind of *discussion forum*: this is a suborganization shared by all organizations, through which they exchange information between each other. From the point of view of proof construction, whenever information is exchanged, we make progress in building the proof tree in the direction of the root; on the other hand, making progress towards the leaves along a certain branch of the proof tree has to do with information which is strictly local to the object identified with the given branch, thus accounting for intraobject communication.

Now, communication in a forum is characterized by two basic kinds of speech acts:

- the act of the speaker's addressing one specific hearer, in front of the remaining part of the audience (*specific communication*);
- the act of the speaker's addressing the whole audience (*generic communication*).

In both cases, communication is achieved through a shared communication medium, even when it is specific; moreover, generic communication is always *fair*, in the sense that no receiver can limit only to her/himself the use of a message which is meant for the whole community. As we shall see, our approach to forum-based interobject communication will support both specific and generic communication, and will maintain the fairness of generic communication. But we shall also ensure the *safeness* and the *privacy* of specific communication, by providing a clean way of generating private names to be used as mail addresses labeling messages which have to be specifically addressed, so that the capability to read one of such messages requires acquaintance of the corresponding mail address; alternatively, from the point of view of the speaker/hearer relationship, we can think of such private names as "interpretation keys", whose acquaintance unlocks the meanings of the messages with which they are associated. Furthermore, we shall characterize the relationship between such a novel form of communication and the well-established blackboard-based communication (see for instance [10]): as it will be shown, forum-based communication can both be viewed as a logical version of blackboard-based communication and also as an operational refinement of it in the sense of being endowed with a more local notion of resource consumption (corresponding to the property of fairness of generic communication) which makes it fully adequate for distributed computing.

We shall also compare our approach to communication with the one adopted for Concurrent Constraint Logic Programming (CCLP) languages described in [24, 15]; indeed, both approaches implement a logic-based form of fair generic communication, and appear as complementing each other in the following sense: in the case of forums the items of information can be removed from the "visual field" of agents (i.e., once accessed they are actually consumed, albeit just within the local scope of the accessing agent); in the case of CCLP languages, items of information are instead permanently stored for all agents.

The remaining part of this paper is organized as follows: Section 2 will give a description of the new computational model of *LO* leading to forum-based communication, and will formally characterize it both in terms of an operational (proof-theoretic) semantics and a model-theoretic semantics; Section 3 will describe two applications of this new model of computation, the first one of which (a graphical application) will be characterized by specific communication, while the other (a concurrent chart parser) will be characterized by generic communication and will provide us with an instance of a highly general methodology for distributed problem solving; Section 4 will compare our approach with related work, and Section 5 will give a brief overview of ongoing directions of research.

2 Description of the Language *LO*

2.1 Formulae, Sequents, Proofs

The syntax of the language *LO* uses three connectives of Linear Logic: "par" (written $\invamp$), "with" (written $\&$), and "top" (written $\top$). We also make use of the Linear implication (written $\multimapinv$) which can be defined in terms of the other connectives of Linear Logic. Two classes of Linear formulae, namely "goals" G and "methods" M, are built recursively from the class A of atomic formulae (i.e. simple terms possibly containing variables), as follows:

$$G = A \mid G \invamp G \mid \top \mid G \& G$$
$$M = A \multimapinv G \mid A \invamp M$$

A "program" is a set of methods and a "context" is a multiset of *ground* goals (i.e. containing no free variables). An *LO* "sequent" is a pair written $\mathcal{P} \vdash \mathcal{C}$ where $\mathcal{P}$ is a program and $\mathcal{C}$ is a context.

2.1.1 Definition of the Inference Figures for *LO*

A proof is a tree structure whose nodes are labeled with sequents. By convention, a proof tree is graphically represented with its root at the bottom and growing upward. Its branches are obtained as instances of the inference figures of the following sequent system, which defines *LO*'s operational semantics.

- Decomposition

$$[\invamp]\ \frac{\mathcal{P} \vdash \mathcal{C}, G_1, G_2}{\mathcal{P} \vdash \mathcal{C}, G_1 \invamp G_2} \qquad [\top]\ \frac{}{\mathcal{P} \vdash \mathcal{C}, \top}$$

$$[\&]\ \frac{\mathcal{P} \vdash \mathcal{C}, G_1 \qquad \mathcal{P} \vdash \mathcal{C}, G_2}{\mathcal{P} \vdash \mathcal{C}, G_1 \& G_2}$$

- Propagation

$$[\multimapinv]\ \frac{\mathcal{P} \vdash \mathcal{C}, G}{\mathcal{P} \vdash \mathcal{C}, A_1, \ldots, A_n}$$

$$\text{if } (A_1 \invamp \cdots \invamp A_n \multimapinv G) \in [\mathcal{P}]$$

In these figures, $\mathcal{P}$ and $\mathcal{C}$ denote, respectively, a program and a context. G, G_1, G_2 denote ground goals and the expression $\mathcal{C}, G$ denotes the context obtained as the multiset union of $\mathcal{C}$ and the singleton G.

In the propagation inference figure [$\multimapinv$], we take $[\mathcal{P}]$ to be the set of all the ground instances of the methods in $\mathcal{P}$. The letters $A_1, \ldots, A_n$ denote ground atoms. Thus, the context in the lower sequent contains (in the sense of multiset inclusion) a submultiset of atoms which matches exactly the head of a ground instance of a method from the program. The upper sequent is obtained by *replacing* in the lower sequent this submultiset with the body of the selected instance of method (i.e. a ground goal).

Notice that, by definition, the elements of a multiset are not ordered. Therefore, the order of the atoms in the head of a method is not relevant.

2.1.2 Operational Interpretation of the Inference Figures

Read bottom-up, a proof gives a static representation (a "snapshot") of the overall dynamic evolution of a system of objects viewed as active processes (agents). Each sequent at the node of a proof-tree encodes the state of an object at a given time. The branches of the proof-tree represent object state transitions.

Thus, the sequent system of *LO* can be interpreted as a general specification of a set of valid object state transitions: the lower sequent in each inference figure is the input state of a valid transition, whose output states (if any) are the upper sequents.

- Inference figure [$\top$], which has no upper sequents, encodes a transition without output states. In other words, it allows termination of objects.
- Inference figure [$\&$] has two upper sequents which share a part of their context. Thus, the two output states of this transition can be viewed as clones, that is, as independent entities with a similar structure. In other words, the connective $\&$ allows creation of objects by cloning.
- Inference figure [$⅋$] aggregates, within the same object, two different components. It allows construction of object states with multiple elements, which lies at the basis of the object/subobject relationship in *LO*.
- Inference figure [$\multimap$] allows the transformation of an aggregation of components (a subobject) within an object.

Notice that, for each transition, the program (left-hand side of the sequents) never changes while the context (right-hand side) is always modified (at least one formula is replaced by another). In other words, the program contains the unrestricted resources of the object, that is, those which can be reused as many times as needed, while any element from the context is a restricted resource, which disappears once used.

LO proofs are characterized by two levels of concurrency: *AND*-concurrency, involving processes evolving on different branches of the proof; and *OR*-concurrency, involving different subprocesses aggregated within a single process, evolving on a single branch. These two forms of concurrency correspond to the two forms of communication which, in the introduction, we have called, respectively, interobject and intraobject communication. The terminology for *AND*/*OR*-concurrency has been chosen to make a direct connection with Linear Logic, where the connective $\&$ responsible for *AND*-concurrency is the (additive) conjunction whereas the connective $⅋$ responsible for *OR*-concurrency is the (multiplicative) disjunction.

2.2 Computational Model

In this paper, we keep the basic computational mechanism already proposed in previous papers, which can be summarized as follows:

$$\boxed{\textbf{Computation} = \textbf{Proof Search}}$$

The important novelty here is in the specification of the class of proofs to be searched, called the target proofs, associated with a given query.

2.2.1 Contextual Proof Search

A query is a pair consisting of a program $\mathcal{P}$ and a ground goal G. Target proofs are then defined as follows:

Definition 1 *A target proof is an LO-proof such that its root is a sequent of the form* $\mathcal{P} \vdash C, G$*, where* C *is a context (also called an answer context for the query).*

In other words, proofs are searched in such a way that the context of their root node may *properly contain* the query goal. This new model of computation can be used within the two different paradigms of transformational and reactive programming [13].

- In the "transformational" paradigm, the systems reads an input, processes it and produces an output. The input is here the initial query and the output is any possible answer context $\mathcal{C}$. The elements of $\mathcal{C}$ can be viewed as constraints and thus, a query can be interpreted as "find a set of constraints from which a given formula is derivable".

- In the "reactive" paradigm, several agents interact together by exchanging messages. There is no notion of input and output in this case; the initial query is used only for the purpose of bringing into life certain agents. The answer context $\mathcal{C}$ acts as a medium of communication between agents. $\mathcal{C}$ is initially unspecified but each agent can read and write in it during a state transition; each time an agent writes in $\mathcal{C}$, the written formula is automatically propagated to all the other agents. This kind of communication we call *forum-based communication*, by viewing a sender agent as a speaker talking in front of an audience gathered in a forum.

In this paper, we focus on the second paradigm, where the answer context is used as a communication medium.

2.2.2 Example

Consider the following propositional *LO* program $\mathcal{P}$:

$$\begin{array}{l} p \parr a \mathrel{\circ\!-} r. \\ q \parr a \parr b \mathrel{\circ\!-} \top. \\ r \parr b \mathrel{\circ\!-} \top. \end{array}$$

The following proof Π (where the program $\mathcal{P}$ is omitted from the left hand side of the sequents) is a possible target proof for the query $\langle \mathcal{P}\ ;\ p \mathbin{\&} q \rangle$.

$$\Pi \;=\; [\&]\,\dfrac{[\circ\!-]\,\dfrac{[\circ\!-]\,\dfrac{[\top]\,\dfrac{}{\vdash \top}}{\vdash b, r}}{\vdash b, a, p} \qquad [\circ\!-]\,\dfrac{[\top]\,\dfrac{}{\vdash \top}}{\vdash b, a, q}}{\vdash b, a,\ \ p \mathbin{\&} q}$$

Thus, the multiset b, a is an answer context for the query above. Let us go into the details of a possible construction of Π. Initially, the search tree is reduced to a single node

$$\Pi_0 \;=\; \vdash \mathcal{C},\ \ p \mathbin{\&} q$$

where $\mathcal{C}$ is a still unspecified context.

1. Inference figure $[\&]$ applies to the single node of Π_0 and expands it to
$$\Pi_1 \;=\; [\&]\,\dfrac{\vdash \mathcal{C}, p \quad \vdash \mathcal{C}, q}{\vdash \mathcal{C},\ \ p \mathbin{\&} q}$$

2. At this point, no inference figure applies without making some assumption on the content of $\mathcal{C}$. For example, if we assume that $\mathcal{C}$ contains a (i.e. $\mathcal{C} = \mathcal{C}', a$), then the first method of $\mathcal{P}$ applies to the leftmost leaf of Π_1 (inference figure $[\circ\!-]$), and yields
$$\Pi_2 \;=\; [\&]\,\dfrac{[\circ\!-]\,\dfrac{\vdash \mathcal{C}', r}{\vdash \mathcal{C}', a, p} \qquad \vdash \mathcal{C}', a, q}{\vdash \mathcal{C}', a,\ \ p \mathbin{\&} q}$$

3. To continue, we need further assumptions on $\mathcal{C}$. For example, if we assume that $\mathcal{C}'$ contains b (i.e. $\mathcal{C}' = \mathcal{C}'', b$), then the second method of $\mathcal{P}$ applies to the rightmost leaf of Π_2 (inference figures $[\circ\!-]$, and then $[\top]$), and yields
$$\Pi_3 \;=\; [\&]\,\dfrac{[\circ\!-]\,\dfrac{\vdash \mathcal{C}'', b, r}{\vdash \mathcal{C}'', b, a, p} \qquad [\circ\!-]\,\dfrac{[\top]\,\dfrac{}{\vdash \mathcal{C}'', \top}}{\vdash \mathcal{C}'', b, a, q}}{\vdash \mathcal{C}'', b, a,\ \ p \mathbin{\&} q}$$

4. Now, the third method of $\mathcal{P}$ applies to the leftmost leaf of Π_3, and yields
$$\Pi_4 = [\&]\,\dfrac{[\circ\!-]\,\dfrac{[\circ\!-]\,\dfrac{[\top]\,\dfrac{}{\vdash \mathcal{C}'', \top}}{\vdash \mathcal{C}'', b, r}}{\vdash \mathcal{C}'', b, a, p} \qquad [\circ\!-]\,\dfrac{[\top]\,\dfrac{}{\vdash \mathcal{C}'', \top}}{\vdash \mathcal{C}'', b, a, q}}{\vdash \mathcal{C}'', b, a,\ \ p \mathbin{\&} q}$$

5. Finally, Π is identified as the instance of Π_4 in which $\mathcal{C}''$ is the empty multiset.

Of course this construction is far from being the only possible one. At each step, we have made several decisions, some of which were arbitrary. Hence the need to define a control strategy.

2.3 Proof Search Control

2.3.1 The "tell" marker

A large amount of non-determinism in proof search is eliminated by the following result, which identifies a complete subset of *LO*-proofs, so that the search procedure can be restricted to proofs in this subset.

Theorem 1 *A sequent $\mathcal{P} \vdash \mathcal{C}$ is derivable in LO if and only if it has a "focusing" proof, i.e. one in which the bottom context in each occurrence of the propagation inference figure* [$\multimapinv$] *contains only atoms.*

This result is a special case of a more general theorem for full Linear Logic (called the "focusing" theorem), stated in [3], and which is in fact stronger: if any of the decomposition inference figures ([$\top$], [&] or [$\parr$]) applies at one node of the proof, then it can deterministically be applied immediately. Therefore, as long as the current context contains a non-atomic goal, the proof search procedure can be made completely deterministic.

However, once the context contains only atoms, and it is therefore time for the propagation inference figure [$\multimapinv$] to be applied, we are faced with a non-deterministic choice which we would like to control. Here the crucial problem is that of selecting an appropriate method from the program. By adopting a blind search strategy, any method could be triggered: for, as long as no restriction is put on the context, it would always be possible to assume that the method's head is entirely contained in the yet unspecified part of the context. We introduce therefore a pragmatic tool which gives the user control on such assumptions. Let ^ be a special symbol, called the "tell" marker, which can be used to prefix any atom in the head of a method. Thus, the first method of program $\mathcal{P}$ of Section 2.2.2 could be marked as follows:

$$p \parr {}^\wedge a \multimapinv r$$

This means that, to apply this method, the atom p (unmarked) must be found in the *already specified* part of the context, while the atom a (marked) must be assumed in the *still unspecified* part of the context. Of course, when triggering the method, both the marked and the unmarked atoms of the head (here p, a) are replaced by the body of the method (here, r alone). Thus, the head of each method is split into two groups of atoms: those (unmarked) which are asked from the context (i.e. from its already specified part) and those (marked) which are told to the context (i.e. to its still unspecified part).

Now consider the program $\mathcal{P}$ of Section 2.2.2 with the following marking:

$$\begin{array}{l} p \parr {}^\wedge a \multimapinv r. \\ q \parr a \parr {}^\wedge b \multimapinv \top. \\ r \parr b \multimapinv \top. \end{array}$$

It is easy to check that the proof construction described in Section 2.2.2 is the only possible one with respect to the marking above.

The interactions between the two branches created at step 1 in the search illustrates the communication mechanism obtained by this use of the markings in the head of the methods: first the left branch sends a message a to the right branch (step 2); then the right branch receives this message a, sends a message b to the left branch and terminates (step 3). Finally, the left branch receives the message b and terminates (step 4). In both send and receive operations, the message is locally consumed by the concerned agent and disappears from its scope, but not from the scope of the other agent(s). There lies the fundamental difference between our forum-based communication and blackboard-based systems (like Linda [10], for instance), where, once an agent consumes a resource, it takes it away globally also for all the other agents. Similarly, the "forum based" communication mechanism differs from the one available in the CCLP languages described in [24, 15], where "told" constraints are never removed from the local "visual field" of an agent.

Clearly, communication of this kind directly depends on the possibility of suspending and resuming computation. Indeed, notice that after step 1, no method applies to the right branch. However, no failure occurs because of this. Instead computation on the right branch gets suspended; resuming it must wait for the transition on the left branch (step 2) to produce the atom (a) needed to trigger a method on the right branch. Thus, in *LO*'s proof theory, the closed-world notion of *failure* characterizing traditional logic programming languages is replaced by the open-world one of *suspension.* Deadlocks may follow from the situation of suspension of all proof processes.

2.3.2 Information Hiding via Variable Instantiation

It has been shown above that the use of the tell marker ^ provides a form of control on the choice of methods. But, once a method has been selected, another kind of choice is required, in determining an instantiation for the variables of the selected method (this problem did not appear in previous examples since the methods contained no variables). Unification is the traditional solution for this problem; however, we opt here for another mechanism, which suits better the proposed computational model. It can be summarized as follows:

- Instantiation of variables occurring in the unmarked atoms of the head is effected by simple pattern matching with the corresponding atoms in the context.
- All the other variables of the method are instantiated with distinct "new" constants, that is, constants which do not appear in the portion of the proof built so far.

Assume for instance that we have a branch of the proof where the current context is given by

$$\mathcal{C}\ ,\ p(a)\ ,\ q\ ,\ r$$

where $\mathcal{C}$ is the still unspecified part of the context, and we want to apply the following method[2].

$$p(X) \parr q \parr {}^\wedge s(X,Y) \multimapinv t(Y)$$

This is possible since the unmarked atoms of the head, namely $p(X), q$, match a submultiset of the already known part of the context, namely $p(a), q$. This matching instantiates the variable X to a. The (only) other variable, Y, is instantiated with some arbitrary new constant, say c. Now, the method is fully instantiated and can be triggered by assuming that the atom $s(a,c)$ is in $\mathcal{C}$ (since this atom is prefixed with the tell marker). Thus $\mathcal{C} = s(a,c), \mathcal{C}'$, and a new node can be added to the proof:

$$[\multimapinv]\ \frac{\vdash\ \mathcal{C}', r, t(c)}{\vdash\ \mathcal{C}', s(a,c), p(a), q, r}$$

This mechanism for variable instantiation provides a clean way to generate new unique identifiers. Such identifiers can then be used as mail addresses for messages to be sent in the specific mode. The fact that each mail address thus created is bound to be different from any other previously or subsequently created ensures the safeness and privacy of specific communication; information items labeled with a given mail address will be hidden from those potential receivers unacquainted with it[3].

2.4 Phase Semantics

It has been shown in [3] that *LO*'s sequent system is sound and complete wrt Linear Logic. More precisely,

Theorem 2 *A sequent $\mathcal{P} \vdash \mathcal{C}$ is derivable in LO if and only if the sequent $\vdash (!\bar{\mathcal{P}})^\perp, \mathcal{C}$ is derivable in Linear Logic, where $\bar{\mathcal{P}}$ is the conjunction (&) of the methods of $\mathcal{P}$ (universally quantified) and ! is the Linear modality "of-course".*

Notice the use of the modality ! to prefix the program $\mathcal{P}$ in its Linear Logic version; this explicitly marks the elements of the program (the methods) as unrestricted resources, which can be used as

[2]We follow the convention of starting variable identifiers with an uppercase letter.

[3]Identifiers of this kind are related to the *eigenvariables* used in proof theory to introduce fresh constants in the proof; eigenvariables have been recently proposed in [21] as a way of adding information hiding to logic programming. The difference is that eigenvariables have just "forward" and, therefore, local scope on the branch of the proof where they are introduced; by contrast our newly created identifiers are propagated back to the root of the proof tree, so they have global scope.

many times as needed, whereas the elements of the context $\mathcal{C}$ (the goals) are instead bounded resources which can be used just once.

Theorem 2 above shows in proof-theoretic terms that LO is a fragment of Linear Logic. But there is also a model-theoretic characterization of this fact, based on the "Phase Semantics" proposed in [11] as an interpretation of Linear Logic; such a characterization applies to the computational model presented here in a particularly perspicuous manner.

Take a phase model $\mathcal{M}$ to be a given set of "phases"; the denotation of a formula F in $\mathcal{M}$, written $\lceil F\rceil_{\mathcal{M}}$, is a "fact" of $\mathcal{M}$, i.e. a subset of the set of phases verifying certain properties[4]. Intuitively, the phases can be viewed as *actions*, and the denotation of F is the set of actions which must alternatively be performed so as to make F true. This provides a constructive, dynamic notion of truth, which can be contrasted with the non-constructive, static truth of Boolean semantics.

Denotations of formulae in phase models satisfy two nice properties, shared with Boolean semantics:

- Compositionality:
  The denotation of a complex (non-atomic) formula depends solely on the denotations of its components; thus, e.g.

  $$\begin{aligned}\lceil F \,\&\, G\rceil &= \lceil F\rceil \cap \lceil G\rceil \\ \lceil F \parr G\rceil &= (\lceil F\rceil^{\perp} \circ \lceil G\rceil^{\perp})^{\perp}\end{aligned}$$

  where $\circ$ and $^{\perp}$ are operators of the phase model[5].

- Soundness and completeness of the proof system:
  A formula is provable if and only if it holds in all models; i.e.

  $$\vdash F \text{ if and only if for all } \mathcal{M},\ \mathcal{M} \models F$$

  $\vdash$ is the provability relation of Linear Logic and $\mathcal{M} \models F$ means that the empty phase belongs to the denotation of F in $\mathcal{M}$.

[4]See [11] for the exact definitions.

[5]See [11] for the exact definitions.

However, the Phase Semantics has another pleasant feature which does not hold in the Boolean case: there is a "canonical" phase model[6] $\mathcal{M}_o$ in which the following property holds.

$$\vdash F \text{ if and only if } \mathcal{M}_o \models F$$

Such a canonical model can be directly connected to the computational model proposed here, since computing a query Q can be viewed as building its denotation $\lceil Q\rceil_{\mathcal{M}_o}$ in the canonical model, by enumerating the elements (phases) of the set $\lceil Q\rceil_{\mathcal{M}_o}$. More precisely, in the canonical model $\mathcal{M}_o$, the phases are the multisets Γ of formulae of Linear Logic, and the denotation of a formula F is given by

$$\lceil F\rceil_{\mathcal{M}_o} \stackrel{\text{def}}{=} \{\Gamma \,/\, \vdash \Gamma, F\}$$

Now, notice that given an LO query $\langle \mathcal{P}; G\rangle$, the computational model proposed in the previous section precisely attempts to enumerate the elements of $\lceil(\,!\bar{\mathcal{P}}) \multimap G\rceil_{\mathcal{M}_o}$. Indeed, for any answer context $\mathcal{C}$ to the query, the following three equivalent properties hold:

(i) $\mathcal{P} \vdash \mathcal{C}, G$ is derivable in LO (by Definition 1 of an answer context).

(ii) $\vdash\ (\,!\bar{\mathcal{P}})^{\perp}, \mathcal{C}, G$ is derivable in Linear Logic (from *(i)*, by application of Theorem 2).

(iii) $\vdash \mathcal{C}, (\,!\bar{\mathcal{P}}) \multimap G$ and hence $\mathcal{C} \in \lceil(\,!\bar{\mathcal{P}}) \multimap G\rceil_{\mathcal{M}_o}$ (from *(ii)* by definition of the Linear implication $\multimap$ and of the canonical denotation).

As a matter of fact, the proof search procedure described in the previous section can only generate atomic phases (i.e. containing only atoms). If the control strategy induced by the use of the tell marker $\wedge$ were ignored, i.e. if all the possible markings and all the possible variable instantiations were allowed for all the program methods, then all the atomic phases of $\lceil(\,!\bar{\mathcal{P}}) \multimap G\rceil_{\mathcal{M}_o}$ would be generated by exploring all the alternatives at each nondeterministic choice in the procedure (with a backtrack mechanism, for instance). This complements

[6]See [11] for the exact definitions.

the soundness result given by $(i) - (iii)$ above with a completeness result of our operational search procedure with respect to the Phase Semantics. From a practical point of view, completeness and computational tractability are however incompatible: by imposing one specific marking upon the methods, the programmer enforces the order in which the atomic phases of $\lceil (!\bar{\mathcal{P}}) \multimap G \rceil_{\mathcal{M}_\circ}$ are enumerated, but, at the same time, enables possible situations of deadlock which preclude some atomic phases ever to be constructed.

3 Applications

We illustrate the expressiveness of the computational model described above by two simple applications. The first one (Section 3.1) gives an example of specific communication, and the second one (Section 3.2) an example of generic communication. From now on we replace the logical symbols $\parr$, $\&$, $\top$ and $\circ\!-$ with, respectively, keyboard typable symbols `@`, `&`, `#t` and `<>-`, which are used in the actual implementation of *LO*.

3.1 Specific Communication: Computer Graphics

We describe here a simple graphical application for manipulating geometrical drawings on a 2-Dimensional display. This example is a modification of the one given in [4], where it was used to illustrate *LO*'s approach to knowledge sharing in terms of intraobject communication, while streams were used for interobject communication; here, we replace streams with the use of the forum as a communication medium, and we stress aspects of interobject communication. The evolution of the system of agents is modeled by the construction of a proof tree as in Section 2.2.2.

There are three kinds of communicating agents: the user (of the drawings), the drawings and the display device. Hence, the query which brings into life such agents is given by the goal

```
user & drawings & display.
```

together with a program containing methods executable by these three agents. The unspecified context which is incrementally specified by searching a target proof for the query acts as the forum for communication between agents. Communication here is specific, in that it will involve one agent specifically addressing other agents by posting to their mail addresses.

We focus here on the behavior of the *drawing* agents. At the moment of its creation, a drawing is represented as a context containing the following components:

```
drawing , noshape , id(S) , center(O)
```

`S` is an identifier used as a mail address for the drawing for the purpose of sending messages to it. `O` is a point of the screen, encoded in the form of a pair of coordinates, specifying the center of the drawing. Initially, we only need one single prototype drawing, with mail address `proto`; such a prototype, located at the center of the screen, is initialized by expanding the `drawings` agent in the query, and can be later cloned to create new drawings. Expansion of the `drawings` top-level agent is obtained via the following method:

```
drawings <>- drawing @ noshape @
             id(proto) @ center(m(0,0)).
```

Cloning is triggered upon reception of a message `dup/2` (with 2 arguments) told to the forum by, say, the user agent: the first argument and the second argument of this message are, respectively, the mail address of the drawing we clone from and the mail address of the newly cloned drawing. Immediately after cloning, the two drawings differ only by their mail addresses; however, from now on, they follow completely independent evolutions. This is achieved by the following cloning method for drawings, which exploits crucially the connective $\&$, like all methods dealing with creations of new agents:

```
drawing @ id(S) @ dup(S,S1) @ ^ack(S) <>-
        drawing @ (id(S) & id(S1)).
```

The sender of the `dup/2` message can ensure uniqueness of the mail address of the new drawing by using the mechanism for generating new identifiers described in Section 2.3.2. Notice also how the atom `ack(S)` is sent back to the forum as a message acknowledging that the requested creation has taken place. This is because, in this application, the order in which messages are processed is important: for instance, cloning or printing an object before or after moving it leads to two different results. Acknowledgement messages take a very simple form in this application, as we assume that there is only one single sender that needs to be acknowledged (the user); in a situation where multiple senders need to be acknowledged, such messages should contain not just the address of the acknowledging agent, but also the "return" address of the original sender, to ensure that they are properly delivered.

Once created by cloning, each drawing agent can be modified. For example, to move a drawing (by a specified amount `D`) we have the following method.

```
drawing @ id(S) @ center(O) @
    move(S,D) @ ^ack(S) <>-
        drawing @ id(S) @ center(O+D).
```

The prototype drawing `proto` has no specific shape, and, therefore, neither have its clones at the time of their creation. Giving shape to such formless entities involves using a method like the following one, which constrains a drawing to be a square with sides of length `A`.

```
drawing @ noshape @ id(S) @
    make_square(S,A) @ ^ack(S) <>-
        drawing @ square @
        id(S) @ side(A).
```

Printing a square is done via the following method.

```
square @ side(A) @ center(O) @ id(S) @
    print(S) @ ^ack(S) @
    ^line(M1,M2) @ ^line(M2,M3) @
    ^line(M3,M4) @ ^line(M4,M1) <>-
        square @ side(A) @
        center(O) @ id(S).
```

The points `M1,M2,M3,M4` are the four vertices of the square. They must be computed from the center `O` and side `A` of the square (for clarity, this computation is omitted here). The four messages `line/2` sent upon triggering of this method correspond to graphical commands to print the four edges of the square and are meant for the `display` agent. Notice that there is no need for the messages to this agent to be ordered (we assume here for simplicity sake that the `display` agent consumes only `line/2` messages, and the order in which lines are printed is irrelevant). Therefore, a drawing object does not need to wait for an acknowledgement to such messages to pursue its activity.

The flow of information is represented in Fig. 1. Agents are represented in square boxes and messages in round boxes (only their topmost functor is displayed). An arrow from an agent to a message (resp. from a message to an agent) means that the agent produces (resp. consumes) the message.

The example of this section illustrates the synchro-
nization mechanism based on a send/acknowledge protocol between agents sharing a common communication medium, the forum. This communication mechanism is more flexible than the usual stream-based one, in that it saves the programmer from the burden involved in stream manipulations (stream merging, explicit interobject connections, etc.).

3.2 Generic Communication: Concurrent Chart Parsing

The example we provide here is a particularly interesting case of distributed problem solving which illustrates well the use of local resource consumption in generic communication. The problem we address specifically is concurrent parsing, a topic which has attracted the interest of several researchers in the object-oriented programming community [23, 30]; on the other hand the problem-solving technique we employ here can be fruitfully generalized to more complex examples, like distributed expert systems operating on highly complex domains, where different experts are required to work independently on shared data, feeding back

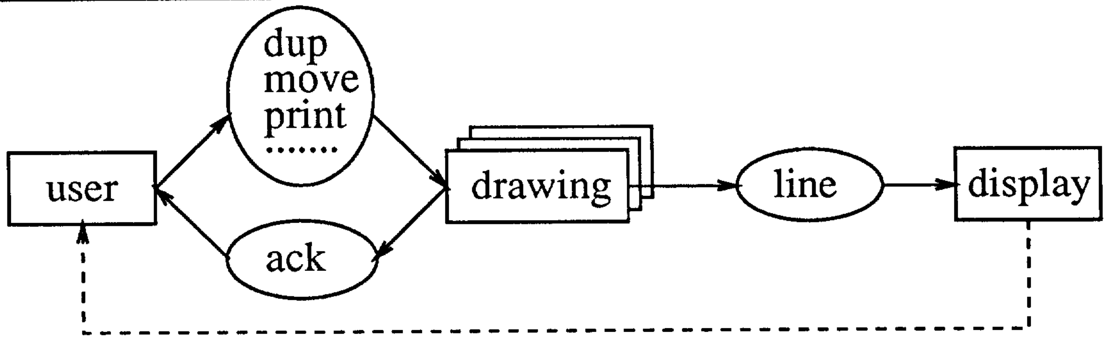

Figure 1: The flow of information

different outputs which all need to be taken in consideration for the final solution of a given problem.

The program we describe amounts to a concurrent implementation of the Earley's algorithm for context-free parsing [9] and draws much in the spirit of the active chart parsing methodology [16], where incomplete phrasal subtrees are viewed as agents consuming already completed elements to produce other (complete or incomplete) subtrees. However, in our case even the rules of the grammar and the entries of the lexicon act as independent units directly partaking in the computation. Moreover, as distinct from the usual sequential formulations of chart parsing, here no superimposed scheduler is in charge of the task of feeding incomplete subtrees with complete ones; instead, incomplete elements behave as truly active decentralized computational units which get their information from the forum, where finished subtrees are told as soon as they have been found. But we must preserve the fact that, once a subtree is completed, this information must be broadcast to *all* the active agents which can make use of it; indeed, in the case of ambiguous grammars, the number of such agents may be greater than one, thus leading to different parses for the same string. Local consumption neatly deals with this problem.

3.2.1 The Program

We view parsing as being performed by four top-level agents, a string scanner, a grammar, a dictionary and a creator of new subtrees. This is expressed by the following method, which contains in its head a single literal `parse(I,S)`, where `I` is the input string and `S` is the symbol of the grammar defining the set of strings with respect to which we want to test membership of `I`.

```
parse(Input,Symbol) <>-
    grammar & dictionary &
    scanner(Input,Symbol) & create_tree.
```

The *scanner* agent, defined in the methods in Fig. 2, performs the two following actions:

- It keeps popping words from the input and producing `pos(N)` and `word(W,N)` messages where
  - a `pos(N)` message supplies the information that position `N` has been reached in the input;
  - a `word(W,N)` message supplies the information that there is a word `W` between positions `N` and `N+1` in the input.

  Positions are encoded as integers in the "successor" notation.
- Upon reaching the end of the input string, it sends a `seek(0,S)` message, where `S` is the targeted grammar symbol, and then reduces itself into an agent whose sole task is that of retrieving answers. This is simply done by waiting for trees covering the whole input string with

```
scanner(I,S) <>-
        scan(I,0) @ target(S).

scan([W|I],N) @ ^pos(N) @ ^word(W,N) <>-
        scan(I,s(N)).

scan([],N) @ target(S) @ ^seek(0,S) <>-
        wait(N,S).

wait(N,S) @ ctree(0,N,S,T) @ ^answer(T) <>-
        wait(N,S).
```

Figure 2: Methods for scanning

```
grammar <>-
        s ==> [np,vp] &
        np ==> [det,n] &
        np ==> [pn] &
        np ==> [np,pp] &
        vp ==> [tv,np] &
        vp ==> [vp,pp] &
        pp ==> [prep,np].
```

```
dictionary <>-
        entry(a,det) &
        entry(robot,n) &
        entry(telescope,n) &
        entry(terry,pn) &
        entry(saw,tv) &
        entry(with,prep).
```

Figure 3: A grammar and a dictionary

```
entry(W,S) @ word(W,N) @ ^ctree(N,s(N),S,S-W) <>-
        entry(W,S).

 (S ==> Ss) @ seek(N,S) @ pos(N) @ ^new(N,N,S,Ss,S) <>-
       (S ==> Ss).
```

Figure 4: Methods for lexical entries and rules

```
create_tree @ new(M,N,S,[],T) @ ^ctree(M,N,S,T) <>-
        create_tree @ ctree(M,N,S,T).

create_tree @ new(M,N,S,[S1|Ss],T) @ ^seek(N,S1) <>-
        create_tree & itree(M,N,S,S1,Ss,T).

itree(M,N,S,S1,Ss,T) @ ctree(N,P,S1,T1) @ ^new(M,P,S,Ss,T-T1) <>-
        itree(M,N,S,S1,Ss,T).
```

Figure 5: Creation and completion of trees

symbol S to appear in the forum; the structure T with which any of such trees has been represented is then explicitly added as an answer.

The *grammar* and the *dictionary* agent expand, respectively, into a set of grammatical rules and of lexical entries, each originating a different agent; a sample dictionary and grammar[7] are given in Fig. 3. Notice that the grammar is an ambiguous one. The behavior of lexical entries and rule agents is defined in terms of the methods in Fig. 4. Lexical entry agents accept as messages words with which they match and send back corresponding *complete* preterminal trees, labeling the given word with a preterminal symbol. On the other hand, rule agents consume `seek(N,S)` messages *together* with `pos(N)` messages, if the sought grammar symbol S corresponds to their own left-hand side symbol; in this case, they issue back a message for the creation of a new agent encoding an *incomplete* (empty) tree. Crucial is here the fact that the consumption by rule agents of `seek/2` messages must be concomitant with the consumption of matching (in the sense of being characterized by the same integer argument) `pos/1` messages; indeed, this correctly ensures that a rule agent can produce no more than one empty incomplete tree for any position of the input string, given that, for any N, it will be able to consume no more than one `pos(N)` message. In this way, we prevent the possibility of infinite loops of the left-recursive kind deriving from rules like the fourth and the sixth one in the grammar of Fig. 3; furthermore, we block the possibility of redundant analyses. This will be illustrated in describing a sample run of the parser further on in this section.

Creating and completing new trees is accounted for in terms of the methods in Fig. 5. The top-level `create_tree` agent consumes messages of the form `new(M,N,S,Ss,T)` where M and N are, respectively, the two string positions spanned by the new tree to be created, S is the root of the tree, Ss is a list of symbols corresponding to the roots of the complete subtrees which are still needed in order to make this tree complete, and T is the representation associated with the tree itself. It then deterministically chooses between the following two actions:

- in case the list Ss is empty, it sends a message `ctree(M,N,S,T)` to signal that a complete tree with root S and representation T has been found between positions M and N;
- in case the list Ss is of the form `[S1|Ss1]`, it sends a message of the form `seek(N,S1)` and then creates an incomplete tree agent of the form `itree(M,N,S,S1,Ss1,T)`.

As for incomplete tree agents of the form `itree(M,N,S,S1,Ss,T)`, they consume complete trees of the form `ctree(N,P,S1,T1)` to produce messages of the form `new(M,P,S,Ss,T-T1)`. Thus, requests for the creation of new trees can come either from rule agents as answers to `seek/2` messages, or from incomplete tree agents; in the former case such requests can be thought of as leading to the formulation of further hypotheses which need to be verified in order to satisfy a certain initial hypothesis (this is known as step of *prediction* in the usual formulations of the Earley algorithm), while in the second case they follow from having progressed "one step" in the verification of a certain hypothesis (this is known as a step of *completion*). Fig. 6 shows the flow of information among the agents. The convention are the same as in the previous section (Fig. 1), except that we also make use of a thicker arrow to explicitly connect the `create_tree` agent with the agents it creates.

3.2.2 A Sample Run

Let us now briefly consider a sample run of the parser. Assuming the grammar and the lexicon in Fig. 3, consider the goal

```
?- parse(
   [terry,saw,a,robot,with,a,telescope],
   s)
```

After running the parser, the following two answers, corresponding to the two parses of the input sentence, will be found in the global context.

[7]The symbol ==> appearing in the grammar rules is not a primitive of *LO* but simply a convenient infix notation for a binary term constructor.

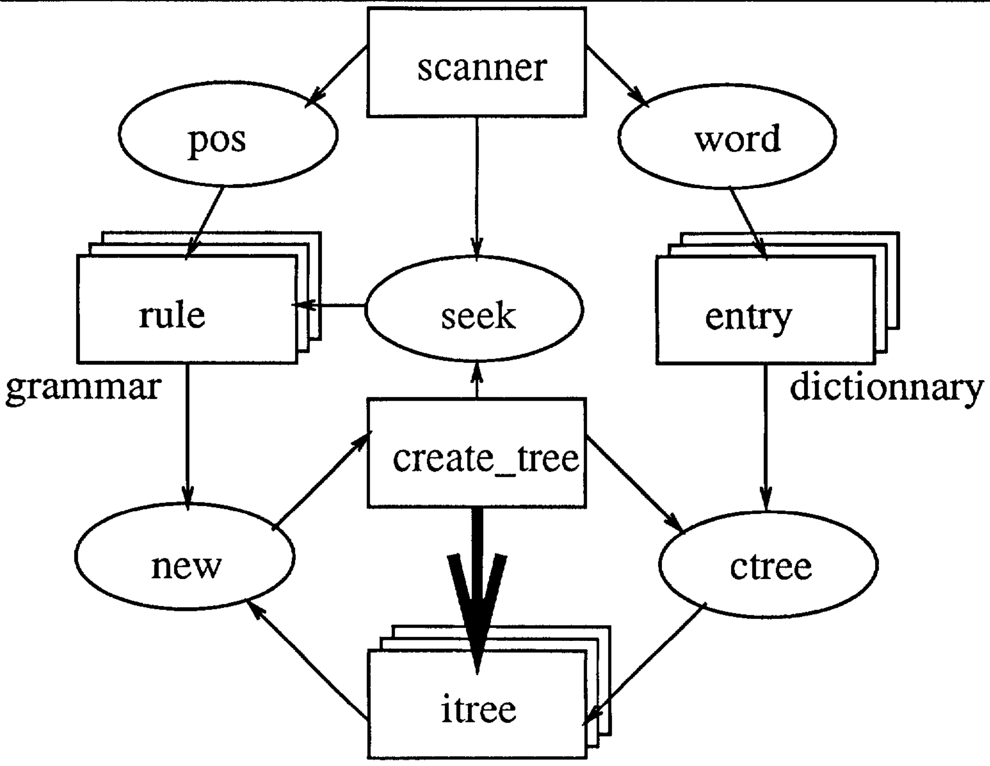

Figure 6: The flow of information

```
answer(
   s-(np-(pn-terry))
    -(vp-(tv-saw)
        -(np-(np-(det-a)-(n-robot))
            -(pp-(prep-with)
                -(np-(det-a)
                    -(n-telescope)))))).
```

```
answer(
   s-(np-(pn-terry))
    -(vp-(vp-(tv-saw)
            -(np-(det-a)-(n-robot)))
        -(pp-(prep-with)
            -(np-(det-a)
                -(n-telescope))))).
```

These two answers originate from the fact that the same complete trees can be consumed by several agents encoding different incomplete trees; specifically, the agents encoded as

```
itree(1,2,vp,np,[],vp-(tv-saw))
```

```
itree(2,2,np,np,[pp],np)
```

will both consume the complete tree

```
ctree(2,4,np,(np-(det-a)-(n-robot)))
```

Furthermore, the agents encoded as

```
itree(1,4,vp,pp,[],
    (vp-(vp-(tv-saw)
       -(np-(det-a)-(n-robot)))))

itree(2,4,np,pp,[],
    (np-(np-(det-a)-(n-robot))))
```

will both consume the complete tree

```
ctree(4,7,pp,
    (pp-(prep-with)
       -(np-(det-a)-(n-telescope))))
```

As a consequence, we end up with two different analyses for the substring *saw a robot with a telescope*. On the other hand, notice that the rules whose left-hand side symbol is np will receive in the course of parsing more than one `seek(2,np)` message to create empty trees with root np and starting position 2; however, any of such rules will never create more than one of such trees, as `seek/2` messages must be consumed together with matching `pos/1` messages, and any rule will be able to consume at most one `pos(2)` message. Thus, both redundant analyses and infinite loops deriving from left-recursion are in this way avoided. This approach to enforcing redundancy checking is quite simple and elegant and comes natural in a decentralized, object-oriented style of programming; it can be contrasted with the more usual way of enforcing it, which is obtained by explicitly comparing newly created trees with previously existing ones.

3.2.3 Summary

We can summarize the salient points of this implementation of a chart parser as follows:

- with respect to sequential implementations, we do not need to take care of specifying a scheduler which handles the feeding of incomplete trees with complete ones;
- with respect to concurrent, stream-based implementations (see for instance [28]) we do not need to bother about the merging of streams of messages coming from different producers;
- with respect to what would be possible in standard blackboard-based communication, we exploit the specific feature of local consumption characterizing forum-based communication, which allows different agents to feed themselves on the same input to produce different outputs.

This produces a concise, "conceptual" style of programming, with little burden on requirements which do not come from the problem itself, but are instead imposed by particular implementation choices. Since the Earley algorithm is an instance of the technique of *dynamic programming*, this approach can be generalized to other examples of dynamic programming, as shown in [5].

4 Related Work

We have seen how forum-based communication, which lies at the basis of the computational model for *LO* presented here, provides a refinement of blackboard-based communication [10] in terms of local consumption. Proposals for a more local form of blackboard-based communication were also presented in [19] in a non-logical setting, sharing our same intent of making use of blackboards in the context of object-oriented programming. [7] provides instead a logical version of blackboard-based communication in its standard global interpretation.

LO can also be seen as an instance of Concurrent Constraint Programming [25], the programming paradigm towards which the concurrent branch of logic programming languages is naturally evolving. (In a nutshell, we can think of Concurrent Constraint Programming as what becomes of logic programming once it is stripped of its obsolete commitments to Classical Logic, minimal Herbrand models, closed-world assumption etc., and computation is explicitly viewed as the interaction of logical agents refining an initial amount of information by incrementally adding new chunks of information, i.e. *constraints.*) Indeed, *LO* can be considered as a Concurrent Constraint language with agents whose point of view of the outside world changes over time: once an agent has seen a piece of the outside landscape (the forum) then it will not see it anymore, unless it copies it explicitly into its own "local" landscape. This can be contrasted with the Concurrent Constraint Logic Programming languages described in [24, 15], where agents never change their point of view with respect to the outside world (the store of constraints). These two ways of implementing concurrent agents clearly complement each other, as they cover different aspects of concurrent problem solving. Merging of the two approaches in a Linear Logic setting could be possible by permitting permanent elements to be added to the forum; such elements would be distinguished from the non-permanent ones in the fact of being marked by Linear Logic modalities which give them explicitly the status of unrestricted resources.

Linear Logic has been exploited to account for concurrency also in [1, 17]; however, the background there is functional programming, instead of logic programming. [14] exploits the intuitionistic (sequential) version of Linear Logic to refine the control mechanisms of sequential logic programs. [20] describes a general framework for "rewriting logics", suitable for accounting for change in a concurrent programming context. [22] approaches the problem of locality of interaction among concurrent subsystems from the point of view of process algebras.

5 Open Problems

Our main effort is currently in the direction of finding an efficient execution model for the language. Indeed, from a practical point of view, *LO* offers challenging but reasonably solvable implementation issues. We currently have a toy interpreter for the language, written in Prolog (with coroutining facilities to simulate concurrency). Selection and access to the methods is one of the main bottleneck of the interpreter; implementations technique used in production systems are currently being explored to overcome this problem [8]. We also think of a compilation process, based on a type-inference mechanism, which would avoid the accumulation of useless messages in object states (a garbage collector could complete the job at runtime). The ultimate compiler should be able to detect cases of specific communication (one-to-one) and implement it as such, that is, without propagating a specifically sent message to the whole forum in such a case, but sending it directly to the intended receiver. Intermediate cases between specific and generic communication, for instance when an object addresses a certain group of objects, could also be given a special treatment.

Acknowledgement

We are grateful to Gerard Comyn and Alexander Herold for helpful comments on this paper. We also thank Nabiel Elshiewy for helpful discussions.

References

[1] S. Abramsky. Computational interpretations of linear logic. Technical report, DOC, Imperial College, London, U.K., 1990.

[2] 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.

[3] J.M. Andreoli. Proposition pour une synthèse des paradigmes de la programmation logique et de la programmation par objets, 1990. Thèse d'Informatique de l'Université de Paris VI (Paris, France).

[4] J.M. Andreoli and R. Pareschi. LO and behold! concurrent structured processes. In *Proc. of OOPSLA/ECOOP'90*, Ottawa, Canada, 1990.

[5] J.M. Andreoli and R. Pareschi. Dynamic programming as multi-agent programming, 1991. *ECOOP'91* workshop on Object-based concurrent computing.

[6] J.M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, To appear, 1991. (Special issue, Selected papers from ICLP'90).

[7] A. Brogi and P. Ciancarini. The concurrent language shared prolog. *ACM Transactions on Programming Languages and Systems*, To appear, 1991.

[8] M. Clemente. Forthcoming Ms Thesis, TU München.

[9] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2), 1970.

[10] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7, 1985.

[11] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50, 1987.

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

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

[14] J.S Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. In *Proc. of LICS'91*, 1991. To appear.

[15] K. Kahn and V.A. Saraswat. Actors as a special case of concurrent constraint logic programming. In *Proc. of OOPSLA/ECOOP'90*, Ottawa, Canada, 1990.

[16] M. Kay. Algorithm schemata and data structure in syntactic processing. Technical report, Xerox Parc, Palo Alto, U.S.A., 1980.

[17] Y. Lafont. Interaction nets. In *Proc. of 17th ACM Symposium on Principles of Programming Languages*, San Francisco, U.S.A., 1990.

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

[19] A. Matsuoka and S. Kawai. Using tuple space communication in distributed object oriented languages. In *Proc. of OOPSLA'88*, San Diego, U.S.A., 1988.

[20] J. Meseguer. A logical theory of concurrent objects. In *Proc. of OOPSLA/ECOOP'90*, 1990.

[21] D. Miller. Lexical scoping as universal quantification. In *Proc. of the 6th International Conference on Logic Programming*, Lisboa, Portugal, 1989.

[22] L. Monteiro and F.C.N. Pereira. A sheaf-theoretic model of concurrency. Technical report, CSLI, Menlo Park, U.S.A., 1986.

[23] C. Numaoka and M. Tokoro. A decentralized parsing method using communicating multiple concurrent objects. In *Proc. of 2nd International Conference of Technology of Object Oriented Languages and Systems*, Paris, France, 1990.

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

[25] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. Technical report, Xerox Parc, Palo Alto, U.S.A., 1990.

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

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

[28] R. Trehan and P.F. Wilk. A parallel chart-parser for the commited choice logic languages. In *Proc. of the 5th International Conference on Logic Programming*, Seattle, U.S.A., 1988.

[29] K. Ueda. *Guarded Horn Clauses.* PhD thesis, Dept of Information Engineering, University of Tokyo, Japan, 1986.

[30] A. Yonezawa and I. Ohsawa. Object-oriented parallel parsing for context-free gramars. In *Proc. of COLING'88*, Budapest, Hungary, 1988.