Electronic Notes in Theoretical Computer Science 4 (1996)

Distributed Logic Objects: A Fragment of Rewriting Logic and its Implementation

Anna Ciampolini[†], Evelina Lamma[†], Paola Mello[‡], Cesare Stefanelli[† 1]

[†] *DEIS, Università di Bologna*
Viale Risorgimento 2, 40136 Bologna, Italy
[‡] *Istituto di Ingegneria, Università di Ferrara*
Via Saragat, 44100 Ferrara, Italy
{*aciampolini,elamma,pmello,cstefanelli*}*@deis.unibo.it*

Abstract

This paper presents a logic language (called *Distributed Logic Objects*, *DLO* for short) that supports objects, messages and inheritance. The operational semantics of the language is given in terms of rewriting rules acting upon the (possibly distributed) state of the system. In this sense, the logic underlying the language is Rewriting Logic. In the paper we discuss the implementation of this language on distributed memory MIMD architectures, and we describe the advantages achieved in terms of flexibility, scalability and load balancing. In more detail, the implementation is obtained by translating logic objects into a concurrent logic language based on multi-head clauses, taking advantage from its distributed implementation on a massively parallel architecture. In the underlying implementation, objects are clusters of processes, objects' state is represented by logical variables, message-passing communication between objects is performed via multi-head clauses, and inheritance is mapped into clause union. Some interesting features such as transparent object migration and intensional messages are easily achieved thanks to the underlying support. In the paper, we also sketch a (direct) distributed implementation supporting the indexing of clauses for single-named methods.

1 Introduction

Several ways of combining object-oriented and logic programming have been proposed to achieve data abstraction, modularity and code reuse. Some proposals have implemented logic objects on stream-based concurrent logic languages (e.g., [15,23,10]), but this choice is not the best for distribution and scalability. Streams, in fact, behave like shared variables and thus introduce a centralization point in the resulting computational model. In particular, stream communication is programmed by having a producer writing messages

[1] We thank the anonymous referees for their useful comments and the CNR coordinated project on integration of logic and objects.

into a difference list, whose head is read by a consumer. To merge multiple streams, a chain of active *merge* processes is needed, thus requiring extra process reductions and lengthening transmission delay.

In the meanwhile, Meseguer proposed a logic theory of concurrent objects in [19] by defining Rewriting Logic. Rewriting Logic is a very general model of concurrency from which many other models can be obtained by specialization. In this logic, rewriting can take place *modulo* an arbitrary set of structural axioms which could be undecidable. This suggests considering subsets of Rewriting Logic to be efficiently implementable.

For instance, the *Maude* language integrates in a very simple and natural manner functional, object-oriented, relational and concurrent programming by supporting term rewriting, graph rewriting and object-oriented rewriting. In particular, the general form of *Maude* rewrite rules "represents communication events in an object-oriented system where it is possible for one, none, or several objects to appear as participants in the left-hand side of rules" [21].

In a later work [20], Meseguer and Winkler introduce a subset of the *Maude* language called *SimpleMaude*. *SimpleMaude* rules involve only (at most) one object and one method in their left-hand side. This is mainly motivated by the need of having an efficient implementation on a wide variety of parallel architectures, ranging from sequential, SIMD, MIMD, and MIMD/SIMD machines (see [18]).

In this paper, we introduce the language of *Distributed Logic Objects* (*DLO*, for short in the following), that is characterized by active, asynchronously executing agents which communicate through message passing. *DLO* can be considered a particular instance of the general theory of Rewriting Logic where only object-oriented rewriting is supported.

As in [21], the approach we consider for the implementation of *DLO* is translation. The idea is to apply program transformation techniques which are semantics-preserving. In this way, we can allow the full generality of *DLO* even if at the expenses of some efficiency. The target language for transforming *DLO* programs is a concurrent logic language (Rose [7]) with multi-head clauses. In Rose, inter-process communication is performed via multi-head clauses as in [12,22], and *AND*-parallel goals do not share variables in order to avoid centralization points. Rose has been implemented on a parallel architecture based on the transputer technology [8] by extending the abstract machine for Prolog [24] with new instructions and data structures supporting distributed unification, process creation and communication, and control of nondeterminism. In the resulting implementation of *DLO*, we map each logic object into a set of Rose goals and clauses, messages between objects into goal invocations, and object names into logic variables. Furthermore, method definitions are translated into Rose clauses and inheritance is obtained through the notion of clause union.

The translation approach is quite effective, and has been used in the past to implement object-oriented systems on top of concurrent logic languages. By translating distributed logic objects into Rose, we obtain a number of distinguishing features. In particular, since local and remote method invoca-

tions are treated in a uniform way, it is possible to move objects at run-time among the nodes of the distributed system, thus allowing a sort of dynamic load balancing. This makes the real implementation scalable with the underlying architecture. Moreover, object names being mapped into logic variables, intensional messages are easily supported.

The major sources of overhead of the resulting implementation are due to the dynamic creation of remote objects and the broadcasting of messages exchanged through the network. In the transformational approach, broadcasting arises because objects are mapped into logic variables and thus this implementation does employ neither the object addresses nor the inheritance structuring for introducing some kind of "indexing" in selecting methods. We discuss how these sources of overhead can be partially reduced by adopting a direct implementation for a subset of the *DLO* language which corresponds to the fragment of Rewriting Logic where at most one object appears as participant in the left-hand side of clauses.

2 Distributed Logic Objects

The language of *Distributed Logic Objects* aims at integrating the deductive capabilities of logic programming with object-oriented features.

A *DLO* class is a set of (guarded) *DLO* clauses, each one serving some method invocations. *DLO* clauses are multi-head (extended) clauses of the kind:
$\langle M_1, \ldots, M_n \rangle, \langle R_1, \ldots, R_k \rangle, \langle S_1, \ldots, S_m \rangle \leftarrow G | O_1 : m_1, \ldots, O_j : m_j, S'_1, \ldots, S'_q.$
where $q \leq m$, $Pred(S'_1, \ldots, S'_q) \subseteq Pred(S_1, \ldots, S_m)$, $|$ is the commit operator (thus introducing don't care non-determinism), and the guard G is a conjunction of system predicates.

The multi-head of a clause is composed of three multisets of atoms, each one enclosed between angle brackets. The first is the set of atoms (Ms) for methods; the second one (Rs) is for *read-only* state variables, i.e., state variables which do not change their values when the clause is applied; the third one (Ss) is for *mutable* state variables, i.e., variables which possibly change their values because of the clause application.

Atomic goals in the body of a clause ($S'_1, \ldots, S'_q$) are used for modifying the state of an object. In particular, a rule with a mutable atom in the head and another atom with the same name in the body is a rule for modifying the state of the object. Thus, state changing is obtained through recursive calls to the state of an object. State variables mentioned in the head of a clause as read-only cannot occur in the body, thus preventing their modification.

The introduction of read-only atoms is novel with respect to other proposals grounding logic objects on multi-head clauses [9,5], and avoids passing the state variables of an object to the reinstating recursive call if they are not changed. This feature is not simply syntactic sugar (as in [19], for instance), but it has been specifically introduced at the lower level of the implementation (see section 4) in order to reduce the number of processes created and messages exchanged. It is worth noting, however, that although the *DLO* lan-

guage provides explicit notation for read-only atoms, this optimization could be automatically done at compile time, by statically analysing the code.

In the body of a clause explicit method invocations occur. A goal of the kind $O : M$ corresponds to sending a message M (which is an atom) to the object instance with name O. *self*-method invocations have the form $self : M$. In order to avoid centralization points, no sharing of variables among parallel atomic goals and messages in the body of a *DLO* clause occurs. Only atoms in the body of a *DLO* clause that are executed sequentially can share variables. To this purpose, we have introduced the sequential operator & to make explicit the sequential composition of atoms in the body of a clause. The logical meaning of the parallel conjunction $p(X), q(X)$ (where X is unbound) in the body of a *DLO* clause is the following: $\exists X\ p(X) \wedge \exists Y\ p(Y)$. In other words, the scope of a variable in a parallel conjunction is the single atomic goal as in [7], provided that the variable is not bound to a ground term. This simplify the underlying computational model and, as a consequence, its distributed implementation.

With regard to the communication mode, it can be either synchronous or asynchronous, depending on the kind of goal composition. In fact, in case of a parallel goal (*i.e.*, belonging to a parallel composition) the communication is asynchronous, while in the other case (*i.e.*, sequential goals) the communication is synchronous.

For a *DLO* clause to fire, all its consumable (respectively, read-only) heads have to unify (resp., match) with some messages sent and some state values of the target object. Moreover, the guard evaluation must succeed. When the clause fires, all the messages and the atoms unified with mutable heads are consumed. Then, during the body execution, new goals are possibly created and new messages sent.

Example 2.1 Let us consider the following example, where we adopt the standard Prolog notation for variables:

```
class point::
      <projx>, <>, <y(Y)> ← true | y(0).
      <projy>, <>, <x(X)> ← true | x(0).
      <trans(Dx,Dy)>, <>, <x(X),y(Y)> ←
                      X1 is X+Dx, Y1 is Y+Dy |
                      x(X1), y(Y1).
      <print>, <x(X),y(Y)>, <> ← true |
                      printer:print_values([X,Y]).
```

It represents the code of class `point` of bi-dimensional points. The first clause projects a point on the x-axis. The second clause projects the target point on the y-axis. The third clause applies a rectilinear translation of vector `(Dx,Dy)` to the target point. Notice that to obtain the state change (e.g., setting to zero the y coordinate of the target point), the state variables of the target point (e.g., `y(Y)`) to be modified by the method (e.g., `projx`) must occur both in the head (as mutable atom) and in the body of the clause. The (recursive) occurrence of the state variables in the body thus plays the role of the *become*

primitive of Actor languages [2,3].
The last clause serves a `print` request by raising, in its turn, a `print_values` request to the `printer` object which is a system object [2]. Notice that the coordinates `X` and `Y` of the target point are simply read in the (multi-)head but not consumed, therefore they do not need to be reinstated in the body of the clause.

Thanks to the intrinsic nondeterminism of logic programming languages, different clauses can be written for the same method. At run-time, the adoption of a committed-choice behavior for clause applications will ensure that only one of the definitions is used to serve a method request. For instance, suppose the following clause is added to the class `point` of example 2.1:

```
<print>, <x(X),y(Y)> ← true |
                        laser_printer:print_values([X,Y]).
```

When a `print` message is sent to a target point, only one of the two definitions (and therefore only one of the two printers) will nondeterministically serve the request.

The committed-choice behavior of *DLO* ensures that at most one clause – among those which modify the state variables of an object – will fire. Thus, mutual exclusion is automatically guaranteed in accessing the mutable object state. On the other hand, if the state does not change (the atoms are read-only in the head of clauses) no synchronization is enforced and thus neither is sequentiality.

Distributed Logic Objects have some powerful features usually not present in procedural object-oriented languages, which are inherited from the underlying logic and the logic porgramming paradigm:

- Input and output parameters for methods are not statically fixed but are determined at run-time by using unification. This feature makes *DLO* methods more reusable and flexible.
- *Intensional messages* can be easily supported. A message of the kind `O:print`, where `O` is an unbound variable, is broadcasted to each object of the system. Furthermore, the syntax can be easily extended in order to support multicasting to all the objects of a class. Let us newly consider example 2.1. A message `class(point,O):print`, where `O` is an unbound variable, would be sent to each object of class `point`. Notice that for each message sent (intensional or not) exactly one object will serve the request, due to the committed-choice behavior of method definitions [3]. For instance, the intensional message `class(point,O):print` is sent to every instance of the `point` class but only the first point that commits will produce the

[2] We suppose that there are some globally available objects representing system devices, identified in the program by Prolog constants.
[3] This avoids to keep intensional messages pending for a long time. In fact, an intensional message cannot be discarded because new objects which may consume the message can be created later. However, in *DLO* this message pending must be performend until the commit, thus normally being limited in time.

printing of its state. This feature can be compared to a particular form of *pattern-directed communication* present in Actor systems [3], where an actor can send a message to a single arbitrary member of a group.

- *Multi-named methods* can be defined in the style of Maude [19]. Multi-named methods can be implemented as multi-head clauses with more than one method name in the head. This kind of clauses express a communication event in which different messages from distinct objects participate and synchronize in order to possibly modify the state of a target object and send new messages. For example, the following multi-named method added to the class `point` of example 2.1:

  `<projx, projy>, < >, <x(X), y(Y)>` ← `true | x(0), y(0).`

  synchronizes two messages (`projx` and `projy`) in order to simultaneously set the value of the coordinates of a point to the origin of the `x` and `y` axis.

DLO classes can be connected into hierarchies in order to favour non-replication of behavior. A *DLO* class can inherit part of its instance specification (state variables and behavior) from more general classes (called superclasses). In the following, we will not focus on inheritance (see [1]).

3 Operational Semantics

DLO operational semantics can be given in accordance with the true concurrent model [11] in a way very similar to that presented in [19]. The key idea is to represent the distributed state as a multiset of object states and messages that evolves by concurrent application of rewriting rules. Thus, this semantic description outlines the concurrent distributed nature of the language.

In particular, the state of the system is denoted by a multiset of couples of type $O : A$ (where O is an object name and A is an atom) representing both messages and object state variables. A (renamed apart) multi-head clause, C, of an object O with the form:
$\langle M_1, \ldots, M_n \rangle, \langle R_1, \ldots, R_k \rangle, \langle S_1, \ldots, S_m \rangle \leftarrow G | O_1 : m_1, \ldots, O_j : m_j, S'_1, \ldots, S'_q.$
where $q \leq m$, and $Pred(S'_1, \ldots, S'_q) \subseteq Pred(S_1, \ldots, S_m)$, is interpreted as a rewriting rule. This rule is triggered by a set of messages sent to O and unifying with the method patterns $M_1, \ldots, M_n$ in the head of C. Moreover, read-only state variables $(R_1, \ldots, R_k)$ in the head of C and mutable state variables $(S_1, \ldots, S_m)$ have to be matched and unified with the current values of the variables of the object O, representing (part of) the current distributed state of the system. Finally, the guard G must be successfully evaluated. The outcome of the application of clause C is that messages unifying with $M_1, \ldots, M_n$ and state variables unifying with $S_1, \ldots, S_m$ disappear, the state of the object O changes according to the structure of the new state variables $S'_1, \ldots, S'_q$, and new messages $(O_1 : m_1, \ldots, O_j : m_j)$ are sent (after the application of the unifying substitution).

The following rewriting rule describes the behavior of the object oriented system when a clause is applied. Let $\|O\|$ denote the code of an object, $\lfloor\ \rfloor$ multisets, and $\cup$, $\setminus$ multiset union and difference respectively. Let C_l, $l =$

$1, \ldots, q$ and $B_j,\ j = 1, \ldots, k$ denote state variables of object O, and $A_p,\ p = 1, \ldots, n$ be some messages sent to O during the computation. We have:

$$\lfloor O : A_1, \ldots, O : A_n, O : C_1, \ldots, O : C_q, \rfloor \cup \lfloor O : B_1, \ldots, O : B_k \rfloor$$
$$\longrightarrow_r$$
$$\lfloor O_1 : m_1, \ldots, O_k : m_k, O : S'_1, \ldots, O : S'_q \rfloor \theta\gamma \cup \lfloor O : B_1, \ldots, O : B_k \rfloor$$

if the following conditions hold:

- $\exists$ a (renamed apart) clause C belonging to the object O ($C \in \|O\|$);
- $\theta = mgu((A_1, \ldots, A_n, B_1, \ldots, B_k, C_1, \ldots, C_q), (M_1, \ldots, M_n, R_1, \ldots, R_k, S_1, \ldots, S_q))$
- $Eval(G\theta) = \gamma$, where $Eval$ denotes the evaluation of the guard G yielding a computed substitution γ.

Notice that the application of substitutions is component-wise. The substitution $\theta\gamma$ is not applied to the atoms $B_1, \ldots, B_k$ to avoid the creation of bindings for their unbound arguments.

Even if the state of the computation is represented by one single multiset, the rewriting rule applies to a subpart of this multiset which contains elements related to a single object. In this respect, each object constitutes in practice a separate context in a way similar to *Linear Objects* [5].

The computation can be defined in terms of applications of the rewriting rules to disjoint subparts of the current state. Concurrency emerges from the fact that more than one rewriting rule is applied at each step of the computation. The condition to be satisfied in order to simultaneously apply several rewriting rules is that their left- hand sides apply to disjoint sets of mutable elements and messages.

The following rule states how the multiset $\mathcal{S}$ representing the current state of the computation changes because of clause application. Let $\mathcal{S}$ be partitioned into disjoint subparts, $X_i,\ i = 1, \ldots, h$, and possibly overlapping subparts, $R_i,\ i = 1, \ldots, h$ (disjoint from X_i). Intuitively, the idea is to permit the parallel application of k clauses (with $k \leq h$) to disjoint subparts of the current state (X_i) which are consumed and rewritten (into subparts Y_i) according to rule $\longrightarrow_r$ and to possibly overlapping subparts of the current state (R_i) which are accessed in read-only mode, and left unchanged by clause application. This behavior is represented more formally by the following rule:

$$\frac{X_i \cup R_i \ \longrightarrow_r \ Y_i \cup R_i \ (i = 1, \ldots, k)}{(\mathcal{S} \ \longrightarrow \ \mathcal{S} \setminus (X_1 \cup \ldots \cup X_k)) \cup (Y_1 \cup \ldots \cup Y_k)}$$

where $R_i, X_i \subseteq \mathcal{S}$ for $i = 1, \ldots, k$.

Mutual exclusion on the mutable state of an object is automatically guaranteed by the above rule which allows the parallel reduction of clauses only if they do not compete for the same data structure in the current state of the computation.

3.1 Relation with Rewriting Logic

DLO clauses can be interpreted as *rewrite rules* [19]. The outlined *DLO* operational semantics, in fact, corresponds to deduction rules of *Rewriting Logic*. In *Rewriting Logic* deduction is performed by concurrent rewriting modulo structural axioms.

Different types of rewriting are usually considered [18]:

- **term rewriting**, where data structures to be rewritten are *terms*;
- **graph rewriting**, where data structures to be rewritten are *labeled graphs*;
- **object-oriented rewriting**, where data structures to be rewritten are *objects* that interact with each other via asynchronous message-passing.

All these forms of rewriting are supported in the *Maude* language, which integrates in a very simple and natural manner functional, object-oriented, relational and concurrent programming.

In *DLO*, instead, we consider only object-oriented rewriting. As shown in [17], when *Rewriting Logic* is used for object-oriented programming, the structural axioms are associativity, commutativity and identity of a multiset union operator that builds up the configuration of objects and messages. These axioms are implicit in our case, since the order of atoms and messages in a *DLO* clause head is, in practice, not relevant and there exists the identity element *true* with respect to composition of elements in a clause head. Furthermore, as for object-oriented systems based on *Rewriting Logic*, we model the state of the computation as a multiset.

It is worth to notice that, differently from (concurrent) term rewriting [13,16] and object-oriented systems based on *Rewriting Logic* [17], we adopt unification instead of matching for consumable atoms occurring in a clause head. In the case of read-only atoms we use a matching algorithm.

Furthermore, differently from *Rewriting Logic* (and term rewriting systems), in our system congruence in rewriting terms is not present. In fact, in *DLO* it does not happen that rewriting applies to a proper subterm. We use standard unification (or matching, in the case of read-only atoms) algorithm for rewriting terms, thus avoiding the sharing of (nested) data between rewriting clauses which can be a problem in parallel distributed implementations of rewriting systems (see [16,18]).

Like in the language *Maude* [19,21], which is based upon conditional rewriting logic, *DLO* clauses can be conditioned via guards. Thus, *DLO* guarded clauses are equivalent to conditional rewriting rules. However, the commit operator introduced in *DLO* is an extra-logical operator. In fact, through this operator computations are made deterministic. This leads to incompleteness of the resulting logic system, but notably simplifies the implementation avoiding the need for exploring all the alternative subparts of the current state that can be rewritten.

Thanks to the committed-choice nature of *DLO*, each element of the current state of the computation will be rewritten by using at most one clause. Therefore, it is not necessary to follow alternative paths originated by the

application of different clauses to rewrite the same element. Notice that each clause application would possibly assign a different value to the variables of rewritten element.

3.2 Forms of Parallelism

DLO operational semantics outlines the potential parallelism present in the language. The interesting feature is that parallelism has not to be explicitly expressed by the programmer but it is implicitly exploited by the underlying support. As many other concurrent logic programming languages, *DLO* parallelism is fine-grained: this usually implies abundance of potential parallelism. The implicit forms of parallelism exploited in *DLO* can be summarized as follows:

- *inter-object parallelism*: object instances (belonging to the same or to different classes) can execute in parallel since they apply to disjoint sets of atoms. This form of parallelism is inherently related to the AND-parallelism of logic programming.
- *intra-object parallelism*: different threads of control can be simultaneously active on the same object. In particular, different methods can be executed in parallel if they do not modify the value of the same state variables, i.e., if they apply to disjoint sets of atoms to be consumed. This is always the case if the object we consider is non-mutable, i.e., all its methods access the object's state variables in read-only mode. In this case, even several applications of the same method for different requests are performed in parallel. If the object we consider is mutable, i.e., some of its methods changes the object's state, the commit operator ensures that only one method at a time changes the state of the object. For example, methods `projx` and `projy` of example 2.1 can be applied in parallel for the same point instance since they do not involve the same variable. The method `trans`, instead, will be executed in mutual exclusion with respect to both `projx` and `projy` since it shares with them part of the mutable state. However, even if two methods cannot be executed in parallel, both multi-head unification and guard evaluation can be performed in parallel. The acceptances of the two invocations of method `trans` and `projx` for an instance of the `point` class are executed in parallel but, after commitment, only one of them will be served.

How to practically support these different forms of parallelism in a distributed system is discussed in section 4.

4 DLO Distributed Implementation

In this section, we describe the main features of the *DLO* implementation on a distributed memory architecture. Distributed memory parallel systems are significantly more problematic than shared memory ones, because of the overhead present when reading and writing nonlocal variables.

The *DLO* programming system is organized into several levels. It allows programs written in the *DLO* language to be compiled and executed on a parallel transputer-based architecture. The distinct parts composing the architectural scheme are:

- The mapping of *DLO* programs into concurrent logic programs. In fact, *DLO* is implemented by following a transformational approach by mapping *DLO* programs into Rose [7] logic programs.
- The run-time environment. The Rose language support consists of a parallel abstract machine which is an extension of the WAM [24]. The parallel abstract machine of the Rose language has been specifically modified to better fit the needs of *DLO* programming, in particluar to support read-only atoms in the head of clauses.
- The physical architecture. It is represented by the MIMD distributed memory architecture, in this case the transputer-based Meiko Computing Surface.

As in [21], the approach we consider for the implementation of *DLO* is translation. The idea is to apply program transformation techniques which are semantics- preserving. In this way, we can allow the full generality of the language even if at the expense of efficiency. The target language for transforming *DLO* programs is a concurrent logic language (Rose [7]) with multi-head clauses. In Rose, inter-process communication is performed via multi-head clauses as in [12,22], and *AND*-parallel goals do not share variables in order to avoid centralization points. Rose has been implemented on a parallel architecture based on the transputer technology [8] by extending the abstract machine for Prolog [24] with new instructions and data structures supporting distributed unification, process creation and communication, and control of non-determinism. In the resulting implementation of *DLO*, we map each logic object into a set of Rose goals and clauses, messages between objects into goal invocations, and object names into logic variables. Furthermore, method definitions are translated into Rose clauses and inheritance is obtained through the notion of clause union.

Example 4.1 Let us consider the class `point` of example 2.1. Its clauses are transformed into the following Rose program P_1:

```
*point(O), projx(O), y(O,Y) ← true | y(O,0).
*point(O), projy(O), x(O,X) ← true | x(O,0).
*point(O), trans(O,Dx,Dy), x(O,X), y(O,Y) ←
                            X1 is X+Dx, Y1 is Y+Dy |  x(O,X1), y(O,Y1).
*point(O), print(O), *x(O,X), *y(O,Y) ← true |
                            print_values(printer,[X,Y]).
```

where * is added for denoting read-only atoms.

Notice that objects are represented by Rose predicates (i.e., the "*class*" predicates and the predicates corresponding to the object state), and state change is still achieved by substituting values for the state variables in the

recursive calls to these predicates. However, notice that if a method simply accesses the state of an object for reading values but not for modifying them (e.g., method `print` in class `point`), the predicates corresponding to the object state in the resulting translation occur only in the head of the corresponding Rose clause, being them read-only.

The translation approach is quite effective, and has been used in the past to implement object-oriented systems on top of concurrent logic languages. By translating distributed logic objects into Rose, we obtain a number of distinguishing features. In particular, since local and remote method invocations are treated in a uniform way, it is possible to move objects at run-time among the nodes of the distributed system, thus allowing dynamic load balancing. This makes the real implementation scalable with the underlying architecture. Moreover, object names being mapped into logic variables, intensional messages are easily supported.

Parallelism and Granularity

The transformational approach supports all the forms of parallelism peculiar to *DLO*. The inter-object parallelism is supported by the parallel execution of Rose *AND* processes: object instances can execute in parallel. With regard to intra-object parallelism, two methods corresponds to two Rose clauses which are executed in parallel (at least after the commit phase), provided that they rewrite disjoint subparts of an object state. Therefore, after the commit phase, both the clauses will be able to proceed and execute the method body in parallel.

The adopted forms of parallelism are fine grained, and can be efficiently supported by the tightly coupled parallel architecture considered. The grain of parallelism and the relative need of collecting parallelism depends on the features of the available architecture. On a loosely coupled architecture (e.g., a network of workstations) an efficient implementation might require a kind of *serialization*, in order to combine multiple processes (allocated on the same processor) into one and replace local message sends with predicate calls in a way very similar to what has been done for Actors [2].

Transparency

DLO objects are transparent with regard to parallelism and location. In fact, when developing a *DLO* application the programmer has not to be aware of the real degree of parallelism exploited. Parallelism is implicit, sequentiality can be made explicit by using the sequential conjunctive operator. Mutual exclusion in accessing the state variables of an object is directly guaranteed by the underlying support provided that consumable atoms in the head of *DLO* clauses are used.

Furthermore, it is not necessary to be aware of the physical location of an object in order to send it a message. In particular, invoking a method of an object residing on a remote node has exactly the same effect as if performed locally, except for a performance penalty. Whenever an invocation is made, the underlying implementation transparently determines the location of the method that has to perform the task required.

Replication

Object code is contained in the class, possibly replicated on several nodes. Each method is handled by a Rose manager process. If the method code is replicated on several nodes then more manager processes exist, one for each copy. Each manager process remains idle until some request is sent to it. When a request for a method is sent, the method acceptance phase is executed in parallel by all the manager processes of the invoked method. Finally, the method will be served by the first manager process that successfully executes the commit phase. Obviously, creating copies of classes on several nodes is quite expensive, since each method request must be dispatched to all the copies. In addition, all copies are expected to perform the same computation, thus introducing an increasing of the global computational load. This overhead is however limited to the method *acceptance* phase until the commit. The advantage is that class code replication leads to the replication of the object control thread, although limited to the method acceptance phase. The acceptance phase can successfully terminate if there is a sufficient degree of replication to provide the requested method on at least one working node, thus achieving a *limited* form of fault tolerance. Notice that this feature does not provide a *complete* form of fault tolerance since the object is not entirely replicated. In fact, the state variables of an object accessed by a method in a consumable way are replicated in each node where a manager process for the method has been created but, after the commit, only one copy of these state variables survives (i.e., that allocated on the same node of the process successfully completing the commit phase). Therefore, object state variables move from node to node depending on the selected methods, determining a migration transparent to the user, and controlled by the commit operator.

Communication

When translating *DLO* programs into Rose, we map objects' names into logic variables and this is a technique used in most implementations of logic objects. In this way, the concept of message sending is quite far from message passing in traditional object-oriented languages. A sender does not really send the message to the receiver, but rather includes the identifier of the receiver in the message and posts the message to a blackboard-like structure (the set of current goals) from which the receiver picks it up by using unification. The resulting communication mechanism is flexible, since no explicit communication pattern has to be established. Intensional messages can be directly supported by using, in messages, logical variables in place of constants for objects identifiers, and exploiting broadcasting. This, however, has the drawback of introducing inefficiencies, and motivated the adoption of a different approach based on a direct implementation, which is presented in the next section.

The overhead deriving from broadcast communication and distributed unification can be also reduced – as pointed out in [6,4] – by applying static analysis techniques based on abstract interpretation. In particular, they can be suitable to avoid some unification operations which are subject to failure

and useless communications.

Discussion
Some attempts have been done in order to implement systems based on Rewriting Logic on special purpose machines (see, for instance, [14]). Our purpose, as in [16], is different since it consists in implementing *DLO* in general purpose parallel machines, in particular MIMD distributed memory parallel architectures like networks of Transputers. Other attempts of implementing concurrent rewriting systems (and in particular the language *SimpleMaude*) have been done also for SIMD and MIMD/SIMD architectures [18].

The first prototype developed has allowed to experiment the expressive power of *DLO* (and Rewriting Logic) and its impact on distribution: some nice features of distributed object-oriented systems such as dynamicity, transparency, migration and dynamic load balancing are directly provided and even enhanced in our system, with no need for a special treatment at support level. In this respect, our work can be considered a concrete attempt to implement Rewriting Logic on an MIMD distributed memory architecture.

First experimental results have shown the viability of the approach and its scalability. We have experimented *DLO* for implementing a computational-intensive object-oriented real application in the field of low-level vision [1]. Nonetheless, the translation approach suffers the overheads due to the high cost of dynamic creation of processes and their scheduling, plus the cost of message broadcasting and the cost of distributed unification.

Broadcasting arises because objects are mapped into logic variables and thus this implementation does employ neither the object addresses nor the inheritance structuring for introducing some kind of "indexing" in selecting methods. In the following, we discuss how these sources of overhead can be partially reduced by adopting a direct implementation for a subset of the *DLO* language with single-named methods only.

5 A Direct Implementation

As pointed out in the previous section, there is an efficiency problem with the translation approach, similar to the one present in the distributed implementation of a blackboard-like structure. The run-time support has to perform multicasting (i.e., sending a message to a selected group of machines) or even broadcasting communications (i.e., sending a message to all machines) even if *DLO* messages are point-to-point.

In [20], Meseguer and Winkler introduced a subset of the *Maude* language called *SimpleMaude*. *SimpleMaude* rules involve only (at most) one object and one method in their left-hand side. This was mainly motivated by the need of having an efficient implementation. Having at most one object in the head of a rule allows to treat object identifiers as first class elements, and associate them with specific addresses in the node where the object is located (see also [18]). Moreover, messages can be sent to the object at the corresponding address, thus avoiding broadcasting. Finally, having at most one message in the head

of a rule allows to introduce indexing on inherited clauses.

In this section we briefly discuss a direct implementation for the subset of *DLO* with single-named methods only. The fragment of Rewriting Logic here considered corresponds, in practice, to that underlying *SimpleMaude*.

Object reification
In order to limit broadcasting, we should represent object identifiers as machine oriented effective address-like entities. This can be obtained by *reification*, i.e., the direct mapping of object identifiers into process identifiers of the run-time support. The sending of messages to the object is performed by posting messages at the corresponding address. The broadcasting is substituted by point-to-point message exchanges.

Indexing on inherited clauses
In order to support some kind of "indexing" in selecting methods, we rely on data structures similar to C++ virtual tables. In particular, we associate with each class C a *class virtual predicate table* where the addresses of the methods of C are stored. Each entry in the table is a method name. Associated with a method, there is the address of the clause defining the method. If a method is defined by several clauses, then more than one address is reported. When classes are linked into hierarchies, inheritance can be implemented by building one virtual predicate table for each class. The skeleton of each table is determined during the compilation.

Discussion
The drawback of avoiding the broadcast of messages is a more complex implementation of intensional messages. Nonetheless, as in [20], one process can be created to handle this kind of messages and to broadcast them to each object in the system.
Moreover, the reification of object identifiers adopted by the direct implementation reduces the transparency of *DLO* objects with respect to both parallelism and location. In fact, the state variables of an object O are still mapped into parallel processes which possibly migrate during the computation, but both the server process associated with O and the manager processes of O's methods are allocated on specific nodes and do not migrate during the computation.

6 Conclusions

We have presented an object-oriented language based on Rewriting Logic, and discussed its features with particular reference to its implementation on a distributed parallel architecture. The implementation has been obtained via translation on top of a concurrent logic language with committed-choice multi-head clauses and restricted *AND*-parallelism. First experimental results have shown the viability of the approach and its scalability. Nonetheless, the translation approach suffers of the overhead due to the high cost of dynamic

process creation, message passing among objects, plus the cost of scheduling them, and the cost of distributed unification. A direct implementation has been also proposed for a subset of the language with single-named methods only.

The first prototype developed has allowed to experiment the expressive power of *DLO* and its impact on distribution: some nice features of distributed object-oriented systems such as dynamicity, transparency, migration and dynamic load balancing are directly provided and even enhanced in our system, with no need for a special treatment at support level. In this respect, our work can be considered a concrete attempt to implement (a subpart of) Rewriting Logic on an MIMD distributed parallel memory architecture.

References

[1] A.Ciampolini, E.Lamma, P.Mello, and C. Stefanelli. Distributed Logic Objects. Technical Report DEIS, DEIS - University of Bologna, 1994.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[3] G. Agha, S. Frolund, W.Y. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel & Distributed Technology*, 1(2):3–20, 1993.

[4] J.M. Andreoli, T. Castagnetti, and R. Pareschi. Abstract Interpretation of Linear Logic Programming. In D. Miller, editor, *Proceedings of IEEE Symposium on Logic Programming ILPS93.* The MIT Press, 1993.

[5] J.M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. In D.H.D. Warren and P. Szeredi, editors, *Proc. Seventh International Conference on Logic Programming*, pages 495–510. The MIT Press, 1990.

[6] M. Bourgois, J.M. Andreoli, and R. Pareschi. Extending Objects with Rules, Composition and Concurrency: The LO Experience. Technical Report TR-92-26, ECRC, 1992.

[7] A. Brogi. AND-parallelism without Shared Variables. In D.H.D. Warren and Peter Szeredi, editors, *Proc. Seventh International Conference on Logic Programming*, pages 306–324. The MIT Press, 1990.

[8] A. Brogi, A.Ciampolini, E.Lamma, and P.Mello. The Implementation of a Distributed Model for Logic Programming based on Multiple-headed Clauses. *Information Processing Letters*, 42:331–338, 1992.

[9] J.S. Conery. Logical objects. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth International Conference on Logic Programming*, pages 420–434. The MIT Press, 1988.

[10] A. Davison. Polka: A Parlog Object-oriented Language. Technical Report Internal report, Dept. of Computing, Imperial College, 1988.

[11] P. Degano and U. Montanari. Concurrent Histories: A Basis for Observing Distributed Systems. *J.CSS*, 34:442–461, 1987.

[12] M. Falaschi, G. Levi, and C. Palamidessi. A Synchronization Logic: Axiomatic and Formal Semantics of Generalized Horn Clauses. *Information and Control*, 60:36–69, 1984.

[13] J. A. Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In R. Keller and J. Fasel, editors, *Proceedings of Graph Reduction Workshop*, volume 279 of *LNCS*, pages 53–93, Santa Fe (NM, USA), 1987. Springer-Verlag.

[14] J. A. Gougen. The rewrite rule machine project. In *Proceedings of the 2nd International Conference on Supercomputing*, 1987.

[15] K. Kahn, E.D. Tribble, M.S. Miller, and D. G.Bobrow. Objects in concurrent logic programming languages. In *Proceedings of OOPSLA-86*. ACM Press, Portland (Oregon), 1986.

[16] Claude Kirchner and Patrick Viry. Implementing parallel rewriting. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, volume 590 of *LNCS*, pages 123–138. Springer-Verlag, 1992.

[17] Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, pages 309–339. DIMACS Series, Vol. 18, American Mathematical Society, 1994.

[18] Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Livio Ricciulli. Compiling rewriting onto SIMD and MIMD/SIMD machines. In *Proceedings of PARLE'94, 6th International Conference on Parallel Architectures and Languages Europe*, pages 37–48. Springer LNCS 817, 1994.

[19] J. Meseguer. A Logical Theory of Concurrent Objects. In *Proceedings of OOPSLA-ECOOP-90*, pages 101–115. ACM Press, 1990s.

[20] J. Meseguer and T. Winkler. Parallel Programming in Maude. Technical Report CSL-91-08, SRI, 1991.

[21] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[22] L. Monteiro. Distributed Logic: A Theory of Distributed Programming in Logic. Technical report, Universidade Nova de Lisboa, 1986.

[23] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1:25–48, 1983.

[24] D.H.D. Warren. An abstract Prolog instruction set. Technical Report TR 309, SRI International, 1983.