

Coordination in *LO*

Jean-Marc Andreoli
Rank Xerox Research Centre, Grenoble, France

Abstract

Coordinating autonomous, possibly heterogeneous agents requires, at least, a flexible communication infrastructure allowing interactions between the coordinated entities (called the “participants”), respecting their autonomy and capable of dealing with their heterogeneity. Shared dataspace communication models, such as Linda [12], fulfill these requirements. In such models, for each coordinated entity, a piece of code interfacing the entity with the shared dataspace must be provided. The advantage of this approach is that each of these interfaces can be written with whatever languages and systems are most suited to the entity which they interface. However, a drawback is that the coordination code must be incorporated inside these interfaces, and none of them, being attached to individual participants, have a global vision of what the coordinated behavior should be. By analogy with an orchestra, one cannot expect the different players to coordinate their activities simply by listening to the output of the rest of the group; a conductor, who has a global vision of the expected coordinated behavior, is needed.

In this paper, we present an overview of the work based on the *LO* coordination model, aiming at simplifying the design of coordinators, seen as “software conductors”. Coordination in *LO* is specified declaratively using rules. On the one hand, rules are well suited to the kind of flexible synchronization constraints required by coordination. On the other hand, they manipulate only high level properties of the participants, abstracting away the complexity due to their heterogeneity. This complexity is transferred into the interfaces in charge of publishing the properties of the participants.

Rules are a powerful programming concept which can be understood in many different ways. Two concrete coordination programming systems have stemmed from the *LO* framework, adopting different views of rules. One, called *ForumTalk* adopts a “re-active” reading of rules, and considers participants as clients to the coordination service. Communication between the clients and the coordination service is purely asynchronous. The other system called the *CLF* adopts, on the contrary, a “pro-active” reading of rules where the participants become servers to the coordination activity. The two systems are presented in detail here.

1 Introduction

Object oriented technology has been successful because it allows to program complex systems by combining reusable software components, encapsulated into objects. Reusability has been widely acknowledged as an important source of programmers’ productivity increase. However, in traditional object systems, the process of combining re-usable objects is itself poorly supported. Indeed, in each situation, a programmer has to design objects of a new kind, whose sole role is to interconnect and combine the functionalities of other objects. It is in general rather difficult to re-use such coordinating objects, because their code heavily relies on the specific configuration and nature of the objects they coordinate. There are two main problems involved here:

- First, coordination deals with high-level units of functionality which may be realized through objects which do not have exactly the same interface. For example, consider the coordination of a text spell-checker, a bibliography database and a drawing tool in order to produce a document with figures and bibliographical references. Several vendors may provide objects for each of these components. But, as long as no standard is established, different vendors will propose different interfaces for the same functionality, even if the differences are not essential. If the coordination code contains calls to these specific interfaces, it cannot be reused when just one component is swapped with another component with the same global functionality, but different details in the interface.

- Second, coordination requires a lot of flexibility in specifying the sequence of actions to perform on each coordinated object. If the sequencing of two actions A and B is not essential, this should appear in the code, so that the same code can be re-used in both situations where A is performed before B or vice-versa. In fact, in general, coordination only imposes high level synchronization constraints on the actions to perform, whereas most object oriented systems, being imperative in nature, tend to enforce the specification of detailed, low-level sequentialization of actions. In the document example above, one may wish to state high level coordination constraints of the form: “If all the figures have been inserted correctly, and all the bibliographical references have been solved successfully, then the printing of the document can start” without specifying in detail the sequence in which the two tasks in the condition are performed.

In this paper, we present the developments which have stemmed from the abstract model of coordination called *LO* [8, 6] in the direction of the design of concrete coordination environments. Section 2 recalls the main principles of the *LO* model. Section 3 presents one instance of the *LO* model exploiting the notion of forum-based broadcast communication. Section 4 presents another instance of the *LO* model, based on a more refined interpretation of the rules. Section 5 concludes the paper.

2 The *LO* coordination model

2.1 Informal Description

A coordination in *LO* is performed by a set of *LO* agents. Each agent has a state, represented by a multiset of “tokens”, and can perform transitions of three kinds:

Transformation : in this case, the state of the agent is simply transformed into a new state by removing some tokens and inserting new tokens.

Duplication : in this case, the agent is cloned, i.e. it becomes two agents with exactly the same state.

Termination : in this case, the agent disappears.

Furthermore, *LO* agents can communicate by broadcast. Broadcasting a token amounts to inserting a copy of that token inside the state of each agent present in the system at the time of the broadcast.

Both transitions and communications are specified by a rule-based program attached to each agent. A program is a set of rules and each agent continuously tries to apply the rules of its program to its state. Each rule application results in transitions and communications in the corresponding agent. The applicability of a rule to an agent in a given state depends only on the rule itself and the state of the agent. No assumption is made about the mechanism which selects applicable rules in order to apply them, except weak fairness, i.e. an applicable rule cannot be ignored forever in the selection process. No assumption is made on how the tokens are implemented, i.e. removed, inserted, duplicated and destroyed.

The *LO* model has been formalized in the framework of Linear Logic programming [15, 3]. According to the Logic Programming paradigm (especially concurrent logic programming [24]), the evolution of the system of agents is captured by the process of proof construction. More precisely, in *LO*, the overall state of a system of agents is represented, at any time, by an incomplete proof in the sequent system of Linear Logic. It is incomplete in the sense that it contains non-logical axioms at the leaves (open nodes), representing the living agents. The internal nodes in the proof represent the past state of the agents, and the arcs represent the past transitions. A transition is represented by the expansion of an open node whereas a communication by broadcast is represented by an insertion of a formula at all the nodes in the proof.

2.2 The Sequent System

We work in a fragment of Linear Logic [3] in which the sequents (one sided sequents as in the Linear Logic tradition) are multisets of formulae of the form $(!\forall\mathcal{P})^\perp, \mathcal{C}$ where \mathcal{P} represents the program of the agent and contains only “program formulae” and \mathcal{C} represents the state of the agents and contains only “token formulae”. For simplicity purpose, the above sequent is written $\mathcal{P} \vdash_{\text{lo}} \mathcal{C}$ (the subscript is often omitted).

We assume that the atomic formulae are partitioned into two *dual* classes, positive and negative formulae, and the token formulae are taken to be the positive atomic formulae (atoms for short). Atoms are built from an initial alphabet of predicate names (with arity) and a domain of values: an atom is of the form $p(a_1, \dots, a_n)$ where p is a predicate of arity n and $(a_i)_{i=1, \dots, n}$ are values from the domain. Atom patterns are formulae of the form $p(t_1, \dots, t_n)$, where p is a predicate of arity n and $(t_i)_{i=1, \dots, n}$ are first order terms, possibly containing variables

and denoting, once instantiated, values of the domain. The class P of program formulae is built from the class A of atom patterns according to the following syntax:

$$\begin{aligned} P &= A \multimap G \mid A \wp P \\ G &= A \mid G \wp G \mid G \& G \mid \top \end{aligned}$$

Notice the use of the intermediate class G , also called “goal formulae”. Using the associativity of \wp , a program formula is of the form

$$A_1 \wp \dots \wp A_r \multimap G$$

where the “head” is the multiset of atom patterns $(A_i)_{i=1,\dots,r}$ with $r \geq 1$ and the “body” is the goal G . We introduce the following mapping, technically needed below: for each goal formula G we define, by induction, the set $\|G\|$ of its “par-components”, where each par-component is a multiset of atom patterns:

$$\begin{aligned} \|A\| &= \{\{A\}\} \\ \|G_1 \wp G_2\| &= \{c_1 \wp c_2 \text{ where } c_1 \in \|G_1\|, c_2 \in \|G_2\|\} \\ \|G_1 \& G_2\| &= \|G_1\| \cup \|G_2\| \\ \|\top\| &= \emptyset \end{aligned}$$

When reduced to the LO fragment, the inference system of Linear logic can be expressed by a unique inference figure:

$$[LO] \frac{\mathcal{P} \vdash \sigma.C_1, \mathcal{C} \quad \dots \quad \mathcal{P} \vdash \sigma.C_n, \mathcal{C}}{\mathcal{P} \vdash \sigma.A_1, \dots, \sigma.A_r, \mathcal{C}}$$

if

- $A_1 \wp \dots \wp A_r \multimap G$ is a program formula in \mathcal{P}
- σ is a ground substitution
- The set of par-components of G is $\|G\| = \{C_1, \dots, C_n\}$

2.3 Proof Expansion and Insertion

To represent agent state transitions and communications, we consider two mechanisms for proof evolution:

Expansion :

Given an open node ν in the proof, the Expansion of node ν consists of finding an instance of the inference figure of LO whose conclusion is the sequent labelling ν , and installing as many nodes as there are premisses in this instance, labeled with the corresponding sequents. The new nodes are declared open whereas ν is closed. Agent duplication and termination are modelled by Expansion steps involving program formulae with, respectively, several and no par-components. Transformation is modelled, at each Expansion step, by the substitution of the head of the applied program formula with the par-components of its body.

Insertion :

The Insertion of an atom a simply consists of adding the atom a at all the nodes in the proof, including, of course, the open nodes. It thus models the broadcast of a to all the living agents of the system.

Clearly, both mechanisms preserve the correctness of the proof. This is obvious for Expansion, since what is added to the proof is by definition an instance of the inference figure of LO . It is also true for Insertion since all the inference figures are preserved by insertion of an atom in both their premisses and conclusion.

The control of the Expansion mechanism is intrinsically achieved by the sequent at the node at which it is applied. Indeed, the Expansion is driven by the constraint that the selected instance of the inference figure of LO must produce a conclusion identical to the sequent of the current node. This means that an agent transition is controlled, but not fully determined, by its current state. Program formulae therefore represent transition rules (or simply rules).

However, the Insertion mechanism has no intrinsic control, since it can be applied to any proof with whatever atom. This means that arbitrary messages could be arbitrarily broadcast at any time. To remedy this problem, the LO model introduces a pragmatic feature to control Insertion:

The atoms in the head of the rules can be prefixed with a special marker, called the broadcast marker \wedge . Insertion is allowed only immediately before an Expansion and only in order to insert the atoms which are marked in the rule used in the Expansion.

For example, the rule $p \wp \wedge q \multimap r$ can be applied to a node ν labeled with a state of the form p, \mathcal{C} in two steps:

1. The atom q is inserted at all the nodes in the proof. This is allowed since this atom is marked in the rule. Node ν is now labeled with p, q, \mathcal{C} .
2. Node ν is then expanded using the same rule, ignoring the broadcast marker, and yields a new node r, \mathcal{C} .

Hence, the above rule specifies two operations executed synchronously: a transition (here a transformation replacing atom p by r in the state) and a communication (here the broadcasting of atom q).

3 The *ForumTalk* system

ForumTalk [4] exploits the *LO* notion of agent and the broadcast mechanism in order to organize coordination. In general, the problem of broadcast in agent systems is that it may be extremely blind, i.e. a broadcast message will reach all the agents in the system whether they are concerned by the message or not. A naive implementation of the *LO* model would therefore inevitably face the problem of “saturation”, where broadcast messages form ever-growing piles of unused messages (garbage) at each agent. This problem has been given special care in the *ForumTalk* system. In *ForumTalk* this problem is alleviated by the use of various “garbage collection” rules.

3.1 Principles

The architecture of a *ForumTalk* application consists of a coordination service, implemented by several servers possibly distributed over the network, and a set of client applications which access this service. Conceptually, the servers collaboratively maintain a big *LO* proof-tree, called the “*ForumTalk* tree”. Initially, the *ForumTalk* tree is reduced to a single open node containing the distinguished atom ω , and residing on a distinguished server called the “session”. Other servers can then join the session. The *ForumTalk* tree then gets built by Expansion and Insertion operations, available through system primitives invoked from the servers themselves. These primitives are:

forum-join :

This primitive registers the server in the session. There is always only one open node of the *ForumTalk* tree residing in the session and containing the atom ω . The effect of the primitive **forum-join** is to expand the tree at that node, as if the following *LO* rule was applied.

$$\omega \circ - \omega \ \& \ \omega_o$$

Thus, two clones are formed: one, containing ω remains in the session and is ready for further Expansion steps controlled by other invocations of the primitive **forum-join**; the other clone contains the distinguished atom ω_o and is created on the server which invoked the primitive **forum-join**. It is assumed that the primitive **forum-join** is disabled on a server once it has been invoked, so that, at any time, on each server, there is only one open node containing the atom ω_o . This node is continuously expanded as if the following rule was applied.

$$\omega_o \ \wp \ u \circ - \omega_o \ \text{for any atom } u$$

This (global) “garbage collection” rule means that no state needs to be memorized at the open node containing ω_o .

agent-start :

This primitive starts an agent on the server where it is invoked. Each agent has an activity of its own and may perform Expansion and Insertion operations in the *ForumTalk* tree. The effect of the primitive **agent-start** is to expand the tree at the node containing ω_o on the server, as if the following *LO* rule was applied.

$$\omega_o \circ - \alpha(a) \ \& \ \omega_o$$

Thus, two clones are formed: one contains ω_o and handles future invocations of the primitive **agent-start**; the other clone contains the distinguished atom $\alpha(a)$ where a is an “agent descriptor” characterizing the agent to start, passed in the arguments of the primitive **agent-start**. There are several kinds of descriptors (detailed below).

Notice that the global “garbage collection” rule is in competition with the “agent creation” rule for the atom ω_o . This means that all the atoms added by an Insertion step *after* the execution of the **agent-start** primitive are accumulated in the branch started by that agent, whereas they are still deleted by the global garbage collection rule in the other branch. Hence, to avoid saturation, each agent must have its own garbage collection strategy.

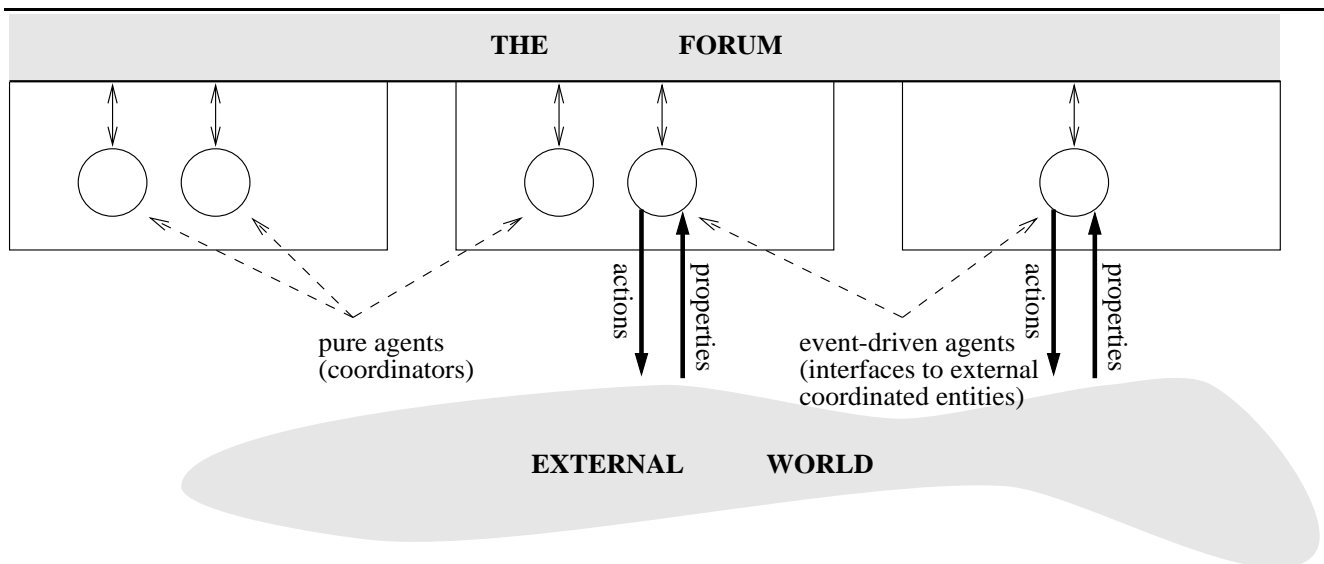


Figure 1: The architecture of a *ForumTalk* application

3.2 Agents Behavior

A *ForumTalk* agent, created by the **agent-start** primitive with a descriptor a , is constrained in the transitions and the communications it can perform:

It is allowed to apply rules only to open nodes which are in the subtree stemming from the original node containing $\alpha(a)$.

This constraint ensures the autonomy of each individual agent. The behavior of an agent is specified by the descriptor a , defining which rules are applied at which open nodes of the subtree of the *ForumTalk* tree stemming from the original node $\alpha(a)$. This decision may depend on (i) the content of the open nodes in the subtree, resulting from the past transitions of the agent but also from communications from other agents, and (ii) events explicitly triggered from the server. Agents relying exclusively on the first policy are called pure, while others are called event-driven. Notice that the initial agents ω and ω_o are examples of event-driven agents.

Pure agents implement the coordination core of a *ForumTalk* application while event driven agents are in charge of the interaction with the external coordinated entities. Typically, the event-driven agents produce atoms representing high-level *properties* of the external world; these atoms are broadcast and processed by the pure agents which in turn produce new atoms representing high-level *actions* on the external entities, which are then broadcast and processed again by the event-driven agents, reflecting these actions on the external world. This architecture is illustrated by Fig. 1. There may of course be several coordinating agents, communicating among themselves by broadcast.

An agent descriptor a consists of a generic agent descriptor a_o which describes the behavior of the agent in terms of generic predicates and a mapping σ (preserving arity) from the generic predicates to the concrete predicates used in the *ForumTalk* tree. In this way, different agents may share the same generic descriptor, and, if they live on the same server, they can share the code implementing this generic descriptor. In the sequel, we write $a = a_o : \sigma$ the agent descriptor obtained from the generic descriptor a_o and the mapping σ .

3.2.1 Descriptors for Pure Agents

Pure agents can be straightforwardly described in *LO* itself. Therefore, we introduce the generic *LO* descriptor of the form $a_o = \tau_{lo}(\mathcal{P}, e)$ where \mathcal{P} is an *LO* program and e a distinguished atom, both being given in terms of generic predicates. The *ForumTalk* agent specified by the descriptor $a = a_o : \sigma$ performs the same Expansion and Insertion operations as allowed by the rules of the *LO* program $\sigma.\mathcal{P}$, plus the initial rule

$$\alpha(a) \circ - \sigma.e$$

Furthermore, the rules of the program \mathcal{P} have a slightly modified syntax w.r.t. pure *LO*: some atoms in the head (left-hand side) of the rules may be of the special form $\{Q\}$ where Q is a request to the underlying server. A request can succeed or fail, and, upon success, may return one or several instantiations for the variables. It is assumed

however that these embedded requests are stateless, i.e. always return the same result whenever they are invoked. One can view embedded requests as special atoms potentially introduced in the state with the initial rule above, and never removed by subsequent application of rules, as if they were infinitely available. For example, the rules can have access to the computational facilities offered by the server, such as arithmetic, as in the following case:

$$p(X) \wp q(Y) \wp \{+(X, Y, Z)\} \circ - p(Z)$$

The special atoms of the form $\{+(a, b, c)\}$ where a, b, c are numbers such that $c = a + b$ are assumed to be potentially infinitely available at all the nodes of *LO* agents in the server.

3.2.2 Descriptors for Event-driven Agents

Several kind of events occurring on the server can be reflected by event-driven agents. Currently, simple input-output events and real time events are handled. The implementation of *ForumTalk* has been designed in an open way to support the adjunction of more sophisticated kinds of event-driven agents.

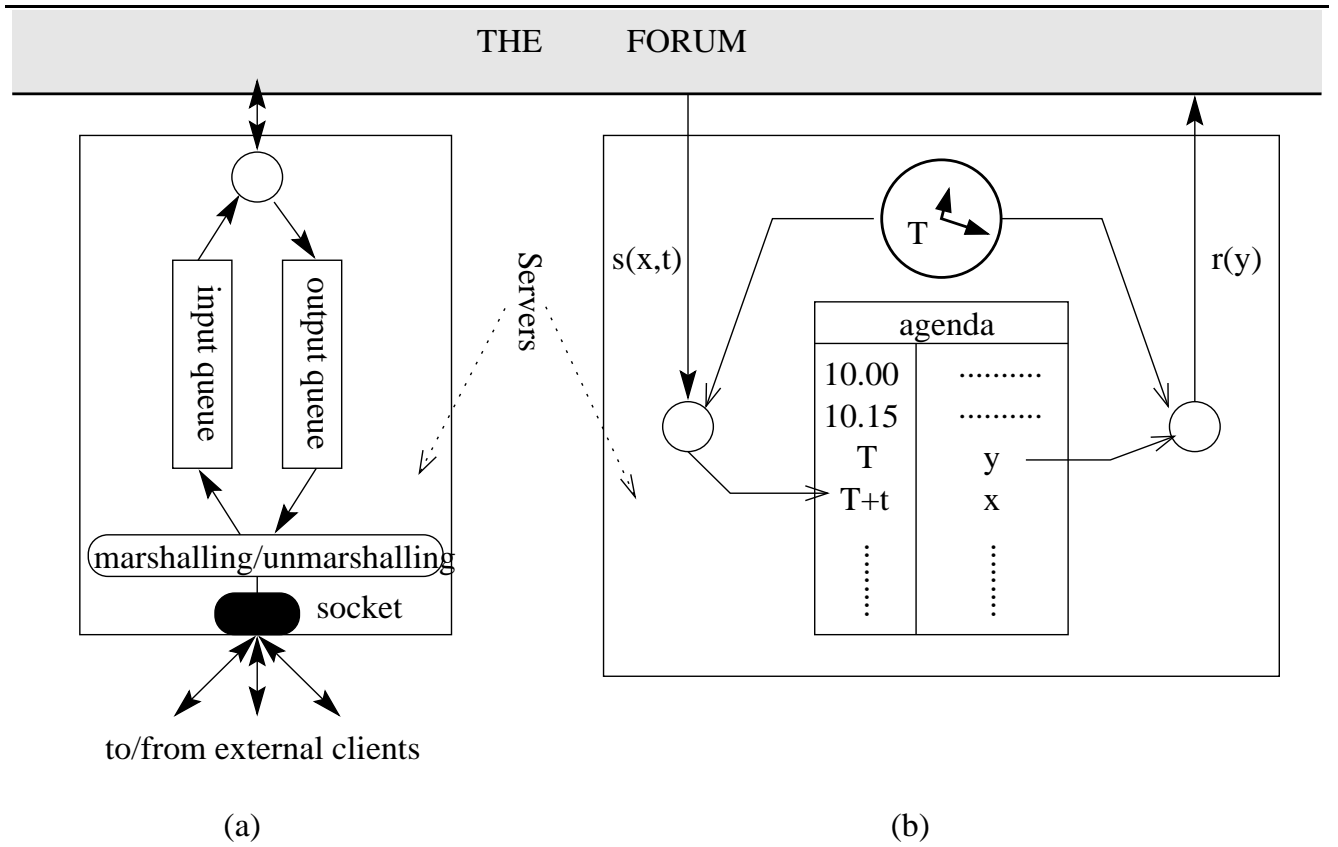


Figure 2: Event driven agents: (a) plug agent (b) agenda agent

Input-Output Events Each server maintains a set of queues, accessible by the client applications via TCP/IP connections. Input queues are filled by client messages read from a set of client sockets; output queues are emptied by writing each message to a set of client sockets. Each queue is constrained by a type specification in an interface definition language called *FDL*. The *FDL* specification attached to a queue describes a set of ground atoms (using generic predicates), called the “support set” of the queue, to which all the elements of the queue must belong. Furthermore, from the *FDL* specification, one can generate the marshalling (resp. unmarshalling) procedures to write (resp. read) the atoms of the queue to (resp. from) the client sockets, according to a specific internal format. It is not the purpose of this paper to describe the syntax of *FDL* nor the format it uses for communication.

A plug (Fig. 2a) is a pair consisting of an input queue and an output queue q_i, q_o , sharing the same set of client sockets, connected to the *ForumTalk* system through an agent with a generic descriptor of the form $a_o = \tau_{\text{plug}}(q_i, q_o)$. The *ForumTalk* agent specified by the descriptor $a = a_o : \sigma$ has the following behavior:

- For any atom u_i in the support set of q_i , the following rule is applicable, whenever u_i is actually in q_i :

$$\alpha(a) \text{ ? } \wedge \sigma.u_i \text{ :- } \alpha(a)$$

and, at the same time, u_i is removed from q_i .

- For any atom u_o in the support set of q_o , the following rule is applicable:

$$\alpha(a) \text{ ? } \sigma.u_o \text{ :- } \alpha(a)$$

and, at the same time, u_o is inserted into q_o .

Client applications can thus asynchronously perform Insertion operations through plug agents and also asynchronously receive notifications of other Insertion operations. The set of clients of a plug is not fixed and can change while the plug is running.

Real Time Events A rudimentary real time service is provided through agenda agents (Fig. 2b) characterized by the generic descriptor $a_o = \tau_{\text{ag}}(g)$ where g is an agenda, i.e. a list of entries consisting of a real time point and a ground term, called an agenda task. We make use here of two distinguished generic predicates, s (arity 2) and r (arity 1). The *ForumTalk* agent specified by the descriptor $a = a_o : \sigma$ performs the Expansion and Insertion operations allowed by the following rules:

- For any ground term x and positive number t , the following rule is applicable:

$$\alpha(a) \text{ ? } \sigma.s(x, t) \text{ :- } \alpha(a)$$

and, at the same time, the agenda g is augmented with an entry consisting of the task x at the time point equal to the current time plus t units.

- For any ground term x and positive number t , the following rule is applicable, whenever there exists in the agenda g an entry consisting of the task x at the time point t and t is greater or equal to the current time:

$$\alpha(a) \text{ ? } \wedge \sigma.r(x) \text{ :- } \alpha(a)$$

and, at the same time, the entry x/t is removed from the agenda g .

In this way, an Insertion operation of an atom of the form $s(x, t)$ is always followed after a delay of *at least* t units of time by an other Insertion operation of an atom of the form $r(x)$.

3.3 Garbage Collection: The Registration Mechanism

An agent descriptor a must specify a “visible set”, i.e. the set of all possible atoms which may be involved in the rules an agent with descriptor a *may* apply during its lifetime. This allows a default garbage collection strategy at the level of each agent. Indeed, when, upon an Insertion, an agent receives a broadcast atom which is not in its visible set, then it can discard this atom immediately. Moreover, whenever communication load is important, the delivery of that message to that agent could even be skipped.

To achieve this, we assume that each atom in the visible set provided in an agent descriptor is marked either as Import, or Export, or both, or none. Export (resp. Import) visible atoms are those which may occur in the head of a rule which may be applied during the lifetime of the agent, prefixed (resp. not prefixed) by the broadcast marker. Let’s consider each of the descriptors introduced in the previous section:

- For an *LO* agent, the visible set consists of all the ground instances of the predicates occurring in the *LO* program (renamed by σ).
- For a plug agent, the visible set consists of the union of the support sets (renamed by σ) of the input and the output queues of the plug. The input (resp. output) support set defines the Import (resp. Export) visible set.
- For an agenda agent, the visible set consists of the atoms of the form $\sigma.s(x, t)$ (Export) and $\sigma.r(x)$ (Import) where x is a ground term and t is a positive number.

The Import visibility information is then used to optimize delivery of broadcast atoms as follows. Each server is equipped with a “forum handler” which acts as local surrogate to the session. When an Insertion operation of an atom u is performed, the atom u is first broadcast to all the agents living on the same server, the assumption being that broadcast is cheap inside the same server. Each of the agents within the server tests whether the atom is in its Import visible set: if it is the case, the agent takes into account the Insertion and propagates it to all its internal nodes, otherwise the atom is discarded for this agent. The local forum handler also receives the atom u and multicasts it to all the forum handlers attached to servers which contain at least one agent having u in its Import visible set, and only those ones. Conversely, when a forum handler receives an atom u from another one, it broadcasts u to all the agents living on its server.

In this way, communication across servers is performed only when needed and each agent processes only the atoms which are in its Import visible set. Of course, the Import visible set does not describe the atoms which *will* be used by the agent, but only those which *may* be used, so that some atoms which reach the agent may still cause saturation. It is then up to each specific agent to provide the garbage collection rules to deal with these remaining atoms. For example, in the case of *LO* agents, we rely on one hand on static analysis tools [5], capable of defining finer grain visible sets per branches inside the *LO* program and on the other hand on explicit, user provided, “garbage collection” rules in the program.

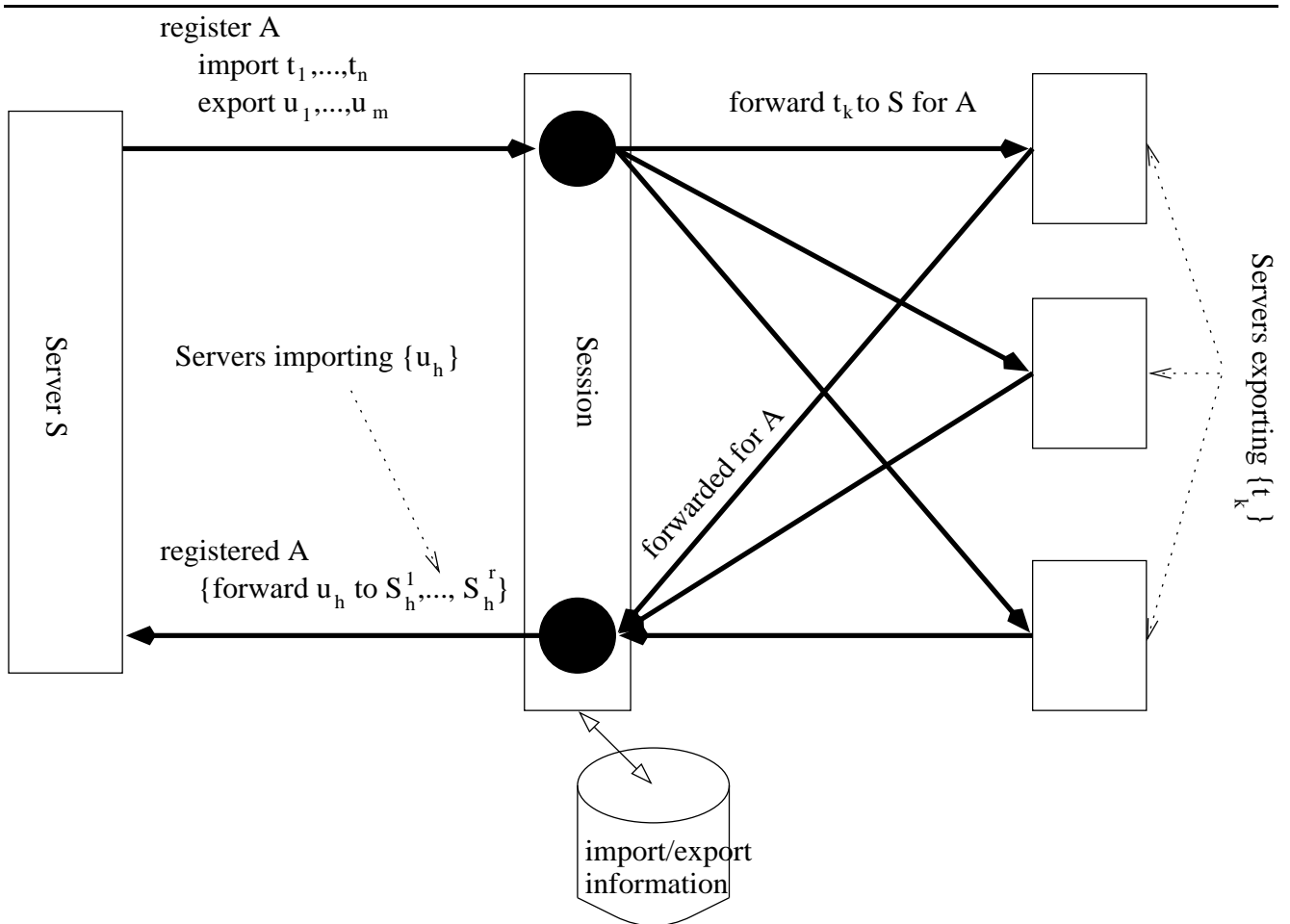


Figure 3: The registration protocol

The previous scheme is possible only if all the forum handlers are continuously informed of the Import visible sets of the agents living on the other servers in the session. This is achieved by requiring that, whenever an agent is started, and before it is actually run, the information about its visible set be broadcast, through the session, to all the other forum handlers. This is called the “registration phase” of the agent, illustrated in Fig. 3. The Export visibility information can be used to optimize the registration phase. Indeed, when a new agent is created, the information about its Import visible set need not be broadcast to all the servers, but only to those which contain an agent whose Export visible set has a non empty intersection with the Import visible set of the created agent. No optimization would be achieved if the Export visibility information also had to be broadcast. Instead, at each

agent creation, both the Import and Export visibility information are passed to the session handler, which is in charge of matching them.

Notice that an agent may cheat about its visibility declaration, and not declare Import atoms which it is sure cannot occur in any export visible set of other agents (or vice-versa). This is the case of atoms which the agent manipulates but wishes to keep private.

In the current implementation, the visibility information is specified in a rather crude way by two lists of predicates, meaning that any ground instance of the first (resp. second) list belongs to the Import (resp. Export) visible set; no constraint can be expressed on the values of the arguments. The two lists may have a non-empty intersection.

4 The *CLF* system

In this section, we present another offspring of the *LO* coordination model, called the *CLF* (Coordination Language Facility). The *CLF* [7] does not crucially rely on the *LO* notion of agent and communication by broadcast, supported by the connective & and the broadcast marker. Furthermore, arguments of predicates in rules are limited to variables instead of full first order terms. In other words, the *CLF* uses very simple production rules, but pays special attention to the way they interact with the outside world. Extension to the full syntax of *LO* rules is possible, but has not been fully investigated yet.

In *ForumTalk*, the rule based coordination core is implemented by the pure agents which interact with the outside world via the event-driven agents. If a message produced by an event-driven agent is interpreted as giving some information about the state of the world, it appears that, due to the inherent asynchrony of the system, this information may be out of date when it is actually consumed and processed by a rule in a pure agent. Conversely, if a message produced by a pure agent is interpreted as requesting an action to be performed on the outside world, asynchrony may cause this action to be actually executed by an event-driven agent too late — or too early. The programmer must be aware of this potential pitfall, and this may, in certain situations, considerably complicate the task of writing of the rules. The *CLF* precisely attempts to tackle this problem.

4.1 Principles

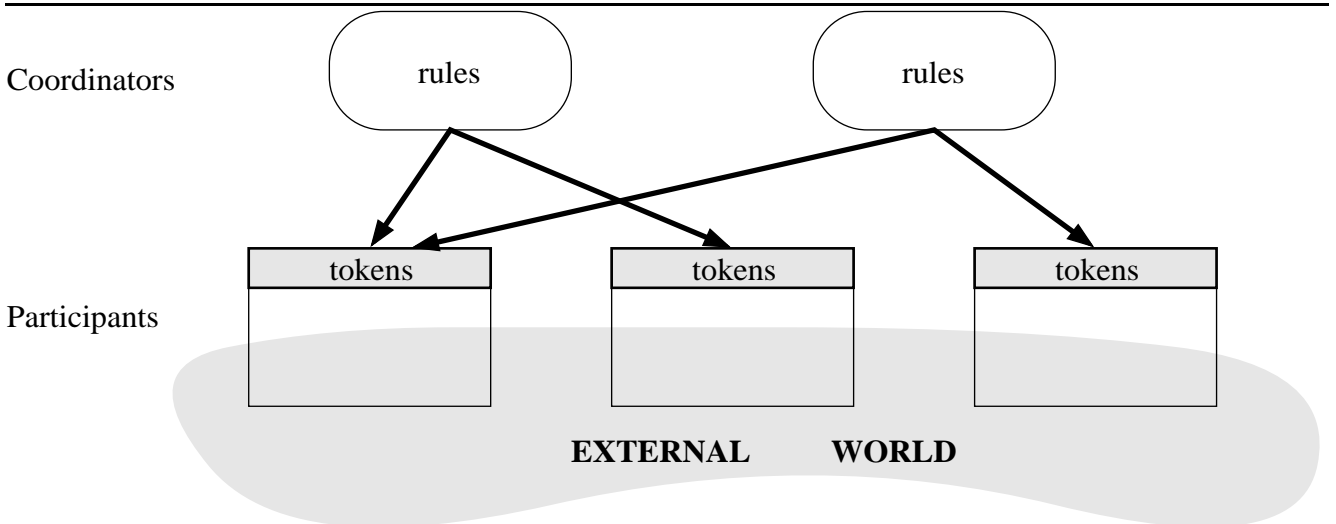


Figure 4: The architecture of a *CLF* application

The architecture of a *CLF* application (Fig. 4) consists of a set of application servers, called the *participants*, which are coordinated by one or several rule based client(s) called the *coordinator*(s). Each coordinator, specified by a set of rules, builds an *LO* proof tree by application of the rules of its program; in fact, given the simplified syntax of the rules, the tree is reduced to a single branch and each arc corresponds to a simple state transformation (multiset rewriting). The originality of the *CLF* resides in the fact that, conceptually, the state of the (single) open node of the tree, called the coordination state, is not handled by the coordinator but is split across the participants. In this way, it is the responsibility of each participant to maintain consistency between its own internal state and the share of the coordination state it handles. Thus, rules always manipulate up-to-date information.

Since the coordinator client does not directly handle the coordination state, it must request it from the participant servers, and must always make sure that the information is valid when it wants to actually use it. This is achieved through a multi-phase protocol.

Inquiry :

In an Inquiry request the coordinator asks the participant to inform it of the presence, at any time, of any atom in the coordination state satisfying a given criterion passed in the argument of the request. Clearly, such a request is “deferred synchronous”, since there may be a number of replies coming asynchronously. Each reply consist of an “action identifier”, characterizing one possible action to perform (on the participant side) to retrieve — and remove — one atom in the coordination state satisfying the requested criterion. More information about the atom may also be passed in the reply.

Reservation :

An action returned in the Inquiry phase may fail when invoked, because, for instance, the atom it was meant to retrieve has disappeared due to a change of state in the participant — and a concomittant change of the coordination state (the participant is in charge of maintaining the consistency between the two states). In order to make sure that the action will not fail, the coordinator must first reserve it. This is the role of the Reserve request, which takes an action as argument and returns either success or failure¹. This request is therefore purely “synchronous”. If it succeeds, then, whenever the coordinator client decides to invoke the action, the participant commits not to fail. If the reservation fails, then the action is garbage collected from the participant and the coordinator is not allowed to invoke it anymore.

Confirmation/Cancellation :

An action which has been successfully reserved can finally be invoked (Confirm request), or given up (Cancel request) by the coordinator. If it is confirmed, the atom in the coordination state that the action was meant to retrieve is actually removed, and the participant is supposed to reflect that change in its own state. If it is cancelled, the participant is relieved from the commitment it took during the reservation and the action is garbage collected. In both cases, no reply is expected from the request, which is therefore purely “asynchronous”. It is an error to attempt to confirm or cancel an action which has not been reserved.

Insertion :

An Insert request takes as argument an atom and inserts it in the coordination state. Again, the participant is supposed to reflect that change in its own state. This request is also asynchronous.

Conceptually, the Reserve/Confirm/Cancel operations of the protocol allow to perform actions returned by various Inquiry operations in a transactional block. This property is used by the coordinators to execute the rules.

4.2 Mapping the Rules to the CLF Protocol

Each rule in a coordinator is implemented as a perpetual thread of activity trying to achieve two goals: (i) instantiate the variables in the rule and (ii) try to apply the ground instances thus obtained. This is achieved by interaction with the participants through the protocol. The instantiation of the variable uses the Inquiry operation while the application of the ground instances uses all the other operations.

4.2.1 Variable Instantiation

As far as variable instantiation is concerned, we make the assumption that the coordinator knows nothing about the domain of values over which the variables range. This knowledge is contained in the participants, and we only assume that each value can be represented (not necessarily in a unique way) in a standard domain (for example strings). The coordinator only manipulates elements of this domain and it is up to the participants to map values into their representation in the standard domain and vice-versa.

Given that the representation of values as strings is not unique, the coordinator cannot even perform identity tests between two values. Therefore, no pattern matching is possible in the variable instantiation process. Instead, the CLF uses a signature driven process. A signature is a declaration, attached to each predicate in the rules of the program, which assigns a role, either Input or Output, to each argument position of the predicate. For instance, the signature

$$p(x, y, z) : x \rightarrow y, z$$

declares that the first argument of predicate p is Input while the second and third arguments of this predicate are output. Notice that nothing prevents signatures with no Input arguments or no Output arguments.

¹There may in fact be other possible answers, detailed later.

The signature declarations are meant to specify the details of the Inquiry operation needed to achieve variable instantiation. The criterion of an Inquiry is given by a predicate and a value (represented as a string) for each Input argument according to the signature of the predicate. Then, each reply to such an Inquiry consists of an action identifier and a value (represented as a string) for each Output argument. In the above example, a correct Inquiry criterion could be given by predicate p with first argument instantiated to a given string x_o . A reply would then consist of an action i and two strings y_o and z_o , for, respectively, the second and third argument of the predicate, meaning that action i is capable of retrieving atom $p(x_o, y_o, z_o)$.

Variable instantiation of a rule is then straightforwardly achieved by sending an Inquiry request for each atom pattern in the head (left-hand side) of the rule. Each request is sent only when its Input arguments have been instantiated by a reply to a previous request. If the rules satisfy a simple correctness criterion, which is tested at compile time, it is easy to see that this mechanism will consistently produce ground instances of the rule, without relying on pattern matching. The correctness criterion states that (i) no variable occurs twice at Output positions of predicates in the head of the rule, (ii) if a variable occurs at an Input position in the head, it also occurs at an Output position in the head, and (iii) if a variable occurs in the body of the rule, it also occurs in the head.

Notice that, since each Inquiry request can return several solutions, the coordinator must maintain all the combinations of solutions of the different requests corresponding to the atom patterns in the head of the rules. This is achieved by an algorithm close to the “Rete” algorithm for production systems.

4.2.2 Application of the Rules

The instantiation mechanism described in the previous section generates ground instances of each rule. A ground instance of a rule consists of an instantiation of its variables by strings, and an assignment of the atoms in the head of the rule to actions returned by the Inquiry operations. To apply a ground instance of a rule, the coordinator performs each of the actions assigned to the atoms in the head of the rule, and then sends Insert requests for each of the atoms in the body of the rule. However, each ground instance of rule must be applied atomically and in isolation, in other words *transactionally*. In absence of any information about the nature of the actions executed in the participants, the *CLF* provides a minimal transaction mechanism.

Atomicity is achieved by a two-phase locking mechanism: first, the coordinator sends a Reserve request for each of the actions assigned to the atoms in the head of the rule; if all the reservations are successful, then it sends Confirm requests for all of them, else it sends Cancel requests for those which have succeeded. Isolation is ensured in a very raw manner: the sequence of reservations for a given rule instance is performed in a critical section, so that no conflict can occur between rule instances inside the same coordinator. However, this method does not prevent reservation conflicts across coordinators. For that purpose, each Reserve operation takes as argument, in addition to the action identifier it is concerned with, a *CLF* “transaction identifier” characterizing the ground rule instance (and the coordinator) which attempts that operation. Now, suppose a reservation has succeeded and a conflicting reservation is attempted in the same participant. The participant is supposed to arbitrate in the following way: if the transaction id of the already successful reservation is greater than that of the attempted one, the special value “busy” is returned; otherwise, the attempted reservation blocks. On the coordinator side, if, in the reservation sequence of a rule instance, none of the reservations fail but some return “busy”, then the whole sequence is aborted and retried. This mechanism ensures that, in case of conflict, the transaction with greater id is performed first in an equivalent serial order. In other words, this mechanism implements a scheduler based on time-stamp ordering. The ordering of the transaction ids is arbitrary, as long as it is total.

The transaction capabilities of the *CLF* are minimal, so that the system can run even without assuming that the participants and the coordinators have access to a common full blown transaction service. However, if such a service is available, nothing prevents the implementation of the operations of the protocol on the participant side to make use of that service. In that case, the *CLF* transaction id can be used to retrieve transaction contexts used in the real transaction system.

4.3 Architecture of a Coordinator

Conceptually, the architecture of a *CLF* coordinator is composed of three modules which correspond to the three operations performed by the rules:

The Negotiation module :

The instantiation of the variables in a rule, based on the Inquiry operation of the protocol, realizes a form of negotiation between the participants: they have to reach an agreement on the values of the variables.

The Transaction module :

The execution of the head of a rule instance, based on the Reserve/Confirm/Cancel operations, realizes a form of transaction.

The Notification module :

The execution of the body of a rule instance, based on the Insert operation, realizes a form of notification that a transaction has been successful.

Variants of the *CLF* can be obtained by tuning the mechanisms which support each of these modules. For example, if a transaction service is available, it can be used in the Transaction module in the coordinator. The Negotiation module can also be fine-tuned. Indeed, one drawback of the signature-based approach is that it does not support bi-directional communication between the participants. Thus, if a participant proposes a value for a variable, and passes it to another participant who is not satisfied with it, then the latter cannot ask the former to propose another value — this kind of backward interaction has to take place outside the system. A more refined Negotiation module would support both directions in the negotiation. A constraint based approach to variable instantiation looks more promising for that purpose.

5 Conclusion and Related Work

In this paper, we have presented two instances of the coordination model *LO*. One, called *ForumTalk*, exploits the broadcast mechanism of *LO* and provides a coordination service with which client applications asynchronously interact. The other, called the *CLF*, concentrates on the interaction between production rules (which is a subset of *LO* rules) and the outside world; the architecture is reversed w.r.t. *ForumTalk*: the *CLF* coordinators are clients which interact at various levels of synchrony with server applications.

A merge of the two approaches may be envisaged in the future. Indeed, it is possible to extend the *CLF* model with the missing connectives and the broadcast mechanism of the *LO* model. In that case, the cloning operation would be interpreted as creating a *version* of a coordination session. Each version of the coordination would handle a separate set of participant versions, one for each participant. Broadcast would enable communication between versions.

Both approaches demonstrate the power of the combination “agents+rules” to design coordination in distributed systems. Agents model in a homogeneous way the possibly heterogeneous, active participants in a coordination. Rules model the behavior of coordinators which interact with the participants through their homogeneous interface.

Several systems have been proposed which use rules to coordinate objects, agents or processes. Some derive from the production rules tradition, and in particular OPS5 [11, 14], while others rather derive from the Prolog tradition [25]. In the production rule approach, let’s mention Neopus [22, 21], which embeds rules in a SmallTalk environment. Commercial rule packages embedded in C++, such as ILOG rules, also exist. In actor languages, such as Actors [1] or ABCL [27], actors coordination can be achieved using filters and synchronizers [2], which behave like production rules managing communication events across actors. Adaptors [26] propose even richer functions to monitor interaction events with rules. In multi-databases environments, active rules [13] have been proposed to coordinate long lived transactions in complex processes. On the side of the Prolog based approaches, several systems have been proposed to unify logic and object-oriented programming, for example [17]. A lot of work has also been done by the database community to provide coordination of database operations based on Prolog rules, for example the VPL system [18], or the deductive object database system [10]. Workflow is yet another area of research which has investigated rules for task coordination, where the coordinated tasks involve any kind of software entities (typically document management systems, text processors, etc.) or even human beings through graphical user interfaces. Most workflow models rely on a Petri-net [23] representation of the flow of tasks [16]. Petri-nets transitions, especially in colored or predicate Petri-nets, are very close to production rules. Finally, rule based coordination has also been investigated in the framework of parallel systems: Gamma [9], or the universal rewrite logic [19, 20] provide models to express the coordination of fine grain activities as found in parallel algorithms. In this case, the rules are rewriting rules which rewrite, in parallel, pieces of abstract structures (multisets or equivalence classes of terms).

Acknowledgement

The author wishes to thank Remo Pareschi and all the members of the “Coordination Technologies” group at RXRC (<http://www.xerox.fr/grenoble/ct/home.html>) for fruitful discussions on the topic of this paper. This work was partly done within the Esprit project 9102 Coordination.

References

- [1] G. Agha and C. Hewitt. Actors: a conceptual foundation for concurrent object-oriented programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, Ma, U.S.A., 1987.
- [2] M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting object interactions using composition filters. In R. Gerraoui, O. Nierstrasz, and M. Riveille, editors, *Object Based Distributed Processing*. Springer Verlag, Berlin, Germany, 1994.
- [3] J-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.
- [4] J-M. Andreoli. Programming in forumtalk. Technical report, Rank Xerox Research Center, Grenoble, France, 1995.
- [5] J-M. Andreoli, T. Castagnetti, and R. Pareschi. Abstract interpretation of linear logic proofs. In *Proc. of ILPS'93*, Vancouver, Canada, 1993.
- [6] J-M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction abstract machines. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 257–280. MIT Press, Cambridge, Ma, U.S.A., 1993.
- [7] J-M. Andreoli, H. Gallaire, and R. Pareschi. Objects meet rules. In *Proc. of Object World Germany'94*, Frankfurt, Germany, 1994.
- [8] J-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Proc. of OOPSLA '91*, Phoenix, Az, U.S.A., 1991.
- [9] J-P. Banâtre and D. Le Metayer. Programming by multiset transformation. *Communication of the ACM*, 36(1), 1993.
- [10] E. Bertino, G. Guerrini, and D. Montesi. Towards deductive object databases. *Theory and Practice of Object Systems*, 1(1):19–39, 1995.
- [11] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS-5*. Addison-Wesley, Reading, Ma, U.S.A., 1985.
- [12] N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge, Ma, U.S.A., 1990.
- [13] U. Dayal, M. Hsu, and R. Ladin. Organizing long running activities with triggers and transactions. In *Proc. of intl. Conf. on Management of Data*, Atlantic City, NJ, U.S.A., 1990.
- [14] C.L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1), 1982.
- [15] J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [16] S. Joosten. Triggering model for workflow analysis. Technical report, Technical University of Twente, Twente, The Netherlands, 1994.
- [17] K. Kahn, E.D. Tribble, M.S. Miller, and D.G. Bobrow. VULCAN: logical concurrent objects. In E. Shapiro, editor, *Concurrent Prolog*. MIT Press, Cambridge, Ma, U.S.A., 1986.
- [18] E. Kühn, F. Puntigam, and A. Elmagarmid. Multi-database transaction and query processing in logic. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, Ca, U.S.A., 1993.
- [19] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 93:73–155, 1992.
- [20] J. Meseguer. A logical theory of concurrent objects and its realization in the MAUDE language. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object Oriented Programming*. MIT Press, Cambridge, Ma, U.S.A., 1992.
- [21] F. Pachet. *Représentation de connaissances par objets et règles : le système NéOpus*. PhD thesis, Université Pierre et Marie Curie, Paris, France, 1992.

- [22] F. Pachet. On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming*, 8(4):19–24, 1995.
- [23] J-L. Peterson. *Petri-Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ, U.S.A., 1981.
- [24] E. Shapiro. The family of concurrent logic programming languages. Technical report, The Weizmann Institute of Science, Rehovot, Israel, 1989.
- [25] E. Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog. *New Generation Computing*, 1(1):25–48, 1983.
- [26] D. Yellin and E. Strom. Interfaces, protocols and the semi-automatic construction of software adaptors. In *Proc. of OOPSLA '94*, Portland, Or, U.S.A., 1994.
- [27] A. Yonezawa, E. Shibayama, Y. Honda, T. Takada, and J-P. Briot. An object-oriented concurrent computation model ABCM/1 and its description language ABCL/1. In A. Yonezawa, editor, *ABCL, an Object-Oriented Concurrent System*, pages 13–45. MIT Press, Cambridge, Ma, U.S.A., 1990.