True Concurrency Semantics for a Linear Logic Programming Language with Broadcast Communication

Jean-Marc Andreoli, Lone Leth, Remo Pareschi and Bent Thomsen

European Computer-Industry Research Center,
Arabellastrasse 17, D-8000 Munich 81 (Germany)
{jeanmarc,lone,bt,remo}@ecrc.de

Abstract. We define a true concurrency semantics for *LO*, a reactive programming language characterized by dynamically reconfigurable agents (processes), with interagent communication implemented as broadcasting and logical operators corresponding to Linear Logic connectives. Our semantic model is given by the well-known Chemical Abstract Machine formalism, where concurrent events happen in the form of chemical-like reactions. Our approach consists of mapping *LO* computations into *CHAM* computations and is easily generalizable to *CHAM*-related models like *CHARMs*, rewriting logics etc. We propose two mappings from *LO* to *CHAMs*, both making use of the "membrane" mechanism of the *CHAM*, but differing in the choice of active elements: in one case, the messages are passive and the agents are the active entities which perform read and write operations; by contrast, in the second case, the agents are passive with respect to communication and the messages themselves move around the solution to deliver their content to each agent. The results in the paper show the effectiveness of the *CHAM* and related formalisms as abstract frameworks for modeling the implementation of practical languages on parallel architectures. Furthermore, they provide insight on the two following issues: (*i*) the amount of synchronization needed to add broadcasting to one-to-one communication primitives; (*ii*) the problem of parallel searching for Linear Logic proofs.
Keywords: True Concurrency, Concurrent Rewriting, Chemical Abstract Machines, Broadcasting, Linear Logic.

1 Introduction

In this paper, we provide a true concurrency semantic characterization for *LO* [AP91b, AP91a, ACP92], a language designed for programming reactive systems. *LO* is finding applications in such fields as parallel algorithms [AP91a, APB91], distributed simulations [ACP92] and as a *coordination* language for extending the capabilities of object-oriented languages [BAP92].

LO views the computation as performed by concurrent agents that are themselves characterized by multiple concurrent internal threads of computation; agents can self-replicate, and their primary form of communication is broadcasting. *LO*'s operators correspond to Linear Logic connectives [Gir87]; hence, *LO* can be viewed as a "linear logic programming language".

One main motivation for giving a true concurrency semantics to *LO* is that it provides us with an exact picture of the different options available for its parallel implementation. More generally, we want also to gain insight on synchronization problems related to broadcasting and on the subject of parallel search of Linear Logic proofs. As far as broadcasting in *LO* is concerned, a major issue is *soundness* (that is, the requirement that every message is delivered exactly once to each receiver), under the assumption that the number of receiving agents may change at run time; this situation is common to other reactive systems, be they concurrent programming languages or operating systems. As far as proofs in Linear Logic are concerned, we shall show that *CHAM*s provide proof encodings characterized by a similar degree of parallelism as Proof Nets [Gir87], but better suited for practical implementation in the context of linear logic programming.

Our model of true concurrency is given by the well-known Chemical Abstract Machine (*CHAM*) framework **[BB90], which extends the Gamma model [BCLM88],** and where concurrent events take the form of chemical-like reactions. Our approach consists of mapping *LO* computations into *CHAM* rewrite sequences and is easily generalizable to formalisms related to the *CHAM* like *CHARM*s [CMR92], rewriting logics [Mes92a] etc.

The rest of this paper is structured as follows. Section 2 recalls the main points about the *CHAM*. Section 3 formally describes *LO* and its model of concurrent computation. In section 4 we show how *LO* computations can be encoded into *CHAM* computations. We start from the basic case of *LO* computations where no broadcasting is involved, and then we consider how things change once broadcasting has to be taken into account; in terms of the corresponding *CHAM* encodings, this will typically imply, with respect to the situation characterized by absence of broadcasting, the introduction of a notion of "time", to indicate that a given agent has already seen, or has not yet seen, a certain message. Different choices will be possible here, by making time either linear or branching; for all cases, theorems will be provided to show the correctness and the completeness of the encodings. Finally, we shall devote section 5 to the discussion of the results.

Remark: due to space restrictions, the proofs of the theorems are omitted. They are available in the technical report version of this paper.

2 *CHAM* Preliminaries

The basic idea of the *CHAM* [BB90] is that the state of a system is like a chemical solution where molecules float around. These molecules can interact with each other according to some reaction rules. Possible contact between molecules is provided for by some means of a stirring mechanism. The solution transformation process is truly parallel since any number of transformations can be performed at the same time, when several rules can be applied to the solution simultaneously and no molecule is involved in more than one rule. Subsolutions can also be transformed in parallel. Solutions are shown using membrane delimiters {| |} and can be treated as molecules or appear as subsolutions in a molecule. In the

following m ranges over molecules and $\mathcal{S}$ ranges over solutions, with $\uplus$ being multiset union.

There are certain laws associated with a *CHAM*. The reaction laws comprise the rewrite rules of the *CHAM*, and these rules are specific to each *CHAM*. The rewrite rules only apply to molecules in solutions, not within molecules. Using the *CHAM* rewrite rules a multiset of molecules is related to a multiset of molecules. A solution consisting of an instance of a left hand side of a reaction rule can be substituted by a solution consisting of the corresponding instance of the rules right hand side. In addition to the reaction laws there are three meta rules [BB90]:

1. The chemical law
$$\frac{\mathcal{S} \longrightarrow \mathcal{S}'}{\mathcal{S} \uplus \mathcal{S}'' \longrightarrow \mathcal{S}' \uplus \mathcal{S}''}$$
describes how parts of a multiset can be rewritten.
2. The membrane law
$$\frac{\mathcal{S} \longrightarrow \mathcal{S}'}{\{\!|\, C[\mathcal{S}] \,|\!\} \longrightarrow \{\!|\, C[\mathcal{S}'] \,|\!\}}$$
states that a solution can be rewritten inside any context.
3. The reversible airlock law, presented in some *CHAM*'s,
$$\overline{\{\!|\, \{m\} \uplus \mathcal{S} \,|\!\} \longleftrightarrow \{\!|\, m \triangleleft \mathcal{S} \,|\!\}}$$
converts any arbitrary solution into a singleton solution, isolating the molecule m. This rule adds a means for introducing some sequentiality into the otherwise extremely parallel *CHAM* model.

Apart from these meta rules a *CHAM* will consist of axioms for transforming molecules on the form $m_1, \ldots, m_p \longrightarrow m'_1, \ldots, m'_q$.

3 *LO*

3.1 Programs, Resources, Proofs

LO programs are built from the following operators: "par" (written $\parr$), "with" (written &), "becomes" (written $\multimap$) and "top" (written $\top$). We assume an initial (possibly infinite) set of atomic formulae A from which we can recursively define two classes of expressions: "resource formulae" R and "program formulae" P.

$$R = A \mid R \parr R \mid R \,\&\, R \mid \top$$
$$P = A \multimap R \mid A \parr P$$

A "program" is a set of program formulae and a "context" is a finite multiset of resource formulae. An *LO* sequent is a pair written as $\mathcal{P} \vdash \mathcal{C}$ where $\mathcal{P}$ is a program and $\mathcal{C}$ is a context.

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

- Decomposition inference figures

$$[\parr]\ \frac{\mathcal{P} \vdash \mathcal{C}, R_1, R_2}{\mathcal{P} \vdash \mathcal{C}, R_1 \parr R_2} \qquad [\top]\ \frac{}{\mathcal{P} \vdash \mathcal{C}, \top} \qquad [\&]\ \frac{\mathcal{P} \vdash \mathcal{C}, R_1 \quad \mathcal{P} \vdash \mathcal{C}, R_2}{\mathcal{P} \vdash \mathcal{C}, R_1 \,\&\, R_2}$$

- Progression inference figure

$$[\circ\!\!-]\ \frac{\mathcal{P} \vdash \mathcal{C}, R}{\mathcal{P} \vdash \mathcal{C}, A_1, \ldots, A_n}\ \text{if}\ (A_1 \parr \cdots \parr A_n \circ\!\!- R) \in \mathcal{P}$$

In these figures, $\mathcal{P}$ and $\mathcal{C}$ denote, respectively, a program and a context. R, R_1, R_2 denote resource formulae and the expression $\mathcal{C}, R$ denotes the context obtained as the multiset union of $\mathcal{C}$ and the singleton R. Notice that, by definition, the elements of a multiset are not ordered. Therefore, the order of the atoms in the left-hand side of a program formula is not relevant.

3.2 Operational Interpretation of the Inference Figures

Read bottom-up, a proof with root sequent $\mathcal{P} \vdash \mathcal{C}$ gives us a static representation (a "snapshot") of the overall evolution of a system of agents working on the initial set of resources $\mathcal{C}$ under program $\mathcal{P}$. Each branch of proof represents the evolution of one agent: the nodes are the agent states while the edges are the state transitions. The open leaves are the agents still living at the time of the snapshot.

Program formulae define the allowed state transitions. They can be thought of as composed of two parts, an *input* part (left-hand side of the symbol $\circ\!\!-$) and an *output* part (right-hand side of the symbol $\circ\!\!-$), implementing, respectively, the consumption and the production of resources from/to the agent's state. However, the operation of producing new resources is here more complex than in standard multiset rewriting, and may in fact involve the creation of new agents, or the termination of existing ones. Indeed, when the output part of a program formula is produced into a context by application of the progression inference figure $[\circ\!\!-]$, it is recursively decomposed by application of the decomposition inference figures, which will either terminate the agent (inference figure $[\top]$) or create a new agent by cloning (inference figure $[\&]$) or simply add new components in the context of the agent (inference figure $[\parr]$).

However, we have not yet explained how broadcast communication among agents is realized: as a matter of fact, this is not done in terms of an explicit inference figure but is instead achieved as a side effect of how proof trees are constructed. We detail the mechanism of proof construction in the next section.

3.3 Proof Construction

A "program call" is a pair $\langle \mathcal{P}; R \rangle$ consisting of an LO program $\mathcal{P}$ and a resource formula R.

Definition 1. A target proof for a program call $\langle \mathcal{P}; R \rangle$ is a proof such that its root is a sequent of the form $\mathcal{P} \vdash \mathcal{C}, R$, where $\mathcal{C}$ is a context containing only atoms.

In other words, target proofs are searched in such a way that the context at their root node may *properly contain* the initial resource of the program call.

We consider two proof construction mechanisms. Let Π be any LO proof.

- Expansion:
  Let ν be an open leaf of Π whose sequent matches the lower sequent of an inference figure. Let Π' be the proof obtained by expanding Π at node ν with branches to new open leaves labeled with the upper sequent(s) of the selected inference figure. We write $\Pi \Longrightarrow_e \Pi'$
- Instantiation:
  Let Π' be the proof obtained by adding an occurrence of a given atom to the context at each node of Π. We write $\Pi \Longrightarrow_i \Pi'$.

Clearly, these proof construction mechanisms are sound in the following sense:

Theorem 2. *If Π is a target proof for a given program call, and $\Pi \Longrightarrow_e \Pi'$ or $\Pi \Longrightarrow_i \Pi'$ then Π' is also a target proof for that program call.*

Definition 3. A proof construction sequence is a sequence of proofs $\Pi_1, \ldots, \Pi_n$ such that

$$\forall k = 1, \ldots, n-1 \quad \begin{cases} \Pi_k \Longrightarrow_e \Pi_{k+1} \\ \text{or} \\ \Pi_k \Longrightarrow_i \Pi_{k+1} \end{cases}$$

The trivial proof Π_o reduced to the single node $\mathcal{P} \vdash R$ is obviously a target proof for the program call $\langle \mathcal{P}; R \rangle$. Hence, by application of theorem 2, so is any proof Π such that there exists a proof construction sequence leading from Π_o to Π. Furthermore, it can be shown that the proofs obtained by this method are all the possible target proofs for the program call, so that the two construction mechanisms introduced above are also complete.

In the agent-oriented computational interpretation of proof construction, an expansion step corresponds to an agent state transition whereas an instantiation step corresponds to a form of communication by broadcasting; indeed, the atom which is added to all the nodes in an instantiation step acts as a message broadcast to all the living agents in the system.

3.4 Control of Broadcast Communication

Unfortunately, the two mechanisms of expansion (i.e. state transition) and instantiation (i.e. broadcast communication) are here completely disconnected: indeed, it can be shown that any expansion step permutes with any instantiation step. In order to allow a form of synchronization between the two mechanisms, required in most applications, we introduce a pragmatic tool which gives the programmer some control over the order of execution of expansion steps and instantiation steps in proof constructions (we loose completeness here).

Let $^\wedge$ be a special symbol, called the "broadcast" marker, which can be used to prefix any atom in the input part of a program formula. Consider then the following program formula:

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

This means that, to apply this program formula in an expansion step using the progression inference figure, the atom p (unmarked) must be found in the context of the selected node, while the atom a is added to this context by performing beforehand an instantiation step adding a to all the nodes of the proof[1]. Except in this situation, no other instantiation steps are allowed. A proof construction sequence satisfying this requirement is called "regular". In the rest of this paper, we consider only regular proof construction sequences, and we take the phrase "proof construction" to mean "regular proof construction".

Definition 4. Let Π, Π' be proofs. We write $\Pi \implies \Pi'$ if there exists a (regular) proof construction sequence from Π to Π'.

Clearly, the relation $\Longrightarrow$ is reflexive transitive. Consider for example the following LO program

$$\{\ p \otimes a \multimap \top\ ;\ q \otimes a \multimap q1\ ;\ r \otimes a \multimap \top\ ;\ s \otimes\ !a \multimap \top\ \}$$

Then, as the reader may easily verify,

$$\Pi_1 \implies \Pi_2 \implies \Pi_3 \implies \Pi_4$$

where Π_1, Π_2, Π_3 and Π_4 are the proofs in Fig. 1. (In the figure, the program is omitted from the left-hand side of the sequents.) The step between Π_2 and Π_3 consists of an instantiation step (broadcasting a), prior to an expansion step using the progression inference figure with the fourth program formula on the rightmost open leaf of Π_2. This instantiation step is indeed allowed by the presence of the broadcast marker in the program formula used.

4 Encoding *LO* Computations into *CHAM* Computations

In this section we show how LO computations can be mapped into corresponding *CHAM* computations. The basic idea is that each open leaf of an LO proof, which represents a living agent of the system, is mapped into a sub-solution in the *CHAM*.

The following definition, which will be exploited later on in the translation schemes, maps a resource formula into the set of multisets of atoms which it yields when decomposed (by application of the Decomposition inference figures).

Definition 5. Let R be a resource formula, we take $||R||$ to be the set of "par-components" of R defined inductively as follows. Each par-component is a multiset of atoms.

[1] Notice the difference here between our broadcast marker and the "tell" mechanism of Concurrent Constraint Logic Programming Languages [Sar89]: in our case, a copy of each broadcast message is delivered to each receiver which can locally consume it; on the other hand, told constraints are added to the global constraint store from where they cannot be removed.

$$\Pi_1 \;=\; \vdash (p \parr q) \,\&\, r \,\&\, s$$

$$\Pi_2 \;=\; [\&]\,\frac{[\parr]\,\dfrac{\vdash p, q}{\vdash p \parr q} \qquad [\&]\,\dfrac{\vdash r \quad \vdash s}{\vdash r \,\&\, s}}{\vdash (p \parr q) \,\&\, r \,\&\, s}$$

$$\Pi_3 \;=\; [\&]\,\frac{[\parr]\,\dfrac{\vdash p, q, a}{\vdash p \parr q, a} \qquad [\&]\,\dfrac{\vdash r, a \quad [\multimap]\,\dfrac{[\top]\,\dfrac{}{\vdash \top}}{\vdash s, a}}{\vdash r \,\&\, s, a}}{\vdash (p \parr q) \,\&\, r \,\&\, s, a}$$

$$\Pi_4 \;=\; [\&]\,\frac{[\parr]\,\dfrac{[\multimap]\,\dfrac{[\top]\,\dfrac{}{\vdash \top, q}}{\vdash p, q, a}}{\vdash p \parr q, a} \qquad [\&]\,\dfrac{[\multimap]\,\dfrac{[\top]\,\dfrac{}{\vdash \top}}{\vdash r, a} \quad [\multimap]\,\dfrac{[\top]\,\dfrac{}{\vdash \top}}{\vdash s, a}}{\vdash r \,\&\, s, a}}{\vdash (p \parr q) \,\&\, r \,\&\, s, a}$$

Fig. 1. A subsequence of a regular proof construction sequence

If	Then
$R = R_1 \parr R_2$	$\|\|R\|\| = \{\alpha_1 \uplus \alpha_2$ s.t. $\alpha_1 \in \|\|R_1\|\|$ and $\alpha_2 \in \|\|R_2\|\|\}$
$R = R_1 \,\&\, R_2$	$\|\|R\|\| = \|\|R_1\|\| \cup \|\|R_2\|\|$
$R = \top$	$\|\|R\|\| = \emptyset$
R is an atom A	$\|\|R\|\| = \{\{A\}\}$

Thus, for example, if R is the resource formula $(p \,\&\, q) \parr r$, then recursive application of the definition above yields two par-components, respectively $\{p, r\}$ and $\{q, r\}$. These are indeed the multisets of resources which replace the resource R in the two branches of proof obtained in its decomposition.

We first give the translation from *LO* to *CHAM* computations in the simple case of programs without occurrences of the broadcast marker, and then, we extend the basic mapping scheme with the broadcast mechanism.

4.1 The Basic Case: No Broadcasting

The *CHAM* used here is simply composed of a set of subsolutions. Each subsolution represents an open node in the proof (i.e. a living agent), and contains a multiset of resource formulae (i.e. the state of the corresponding agent).

Definition 6. Let Π be a proof. We take $\textsc{cham}(\Pi)$ to be the *CHAM* solution consisting of the subsolutions of the form $\{\!|\, \mathcal{C} \,|\!\}$ where $\mathcal{C}$ is the context at an open leaf of Π.

The following definitions provide translation schemes for mapping progression and decomposition inference figures into *CHAM* rules in the case of *LO* programs where no program formula is decorated with the broadcast marker $\hat{}$.

Definition 7 Progression rules. Let $P = A_1 \parr \cdots \parr A_r \multimap R$ be a program formula; we take $\overline{P}$ to be the following *CHAM* rule:

$$A_1 , \ldots , A_r \longrightarrow R$$

Thus, according to this definition, the progression inference figure [$\multimap$] disappears altogether in the *CHAM* formulation, as the application of each program formula according to this inference figure gets directly compiled into a corresponding *CHAM* rule.

Definition 8 Decomposition rules. Let R be a resource formula; we take $\overline{R}$ to be the following *CHAM* rule:

If	Then $\overline{R}$ is the rule
$\|\|R\|\| = \emptyset$	$\{\!\| R \lhd \mathcal{S} \|\!\} \longrightarrow .$
$\|\|R\|\| = \{\alpha\}$	$R \longrightarrow \alpha$
$\|\|R\|\| = \{\alpha_1, \ldots, \alpha_n\}$ with $n \geq 2$	$\{\!\| R \lhd \mathcal{S} \|\!\} \longrightarrow \{\!\| \mathcal{S} \uplus \alpha_1 \|\!\} , \ldots , \{\!\| \mathcal{S} \uplus \alpha_n \|\!\}$

Just as above, the decomposition inference figures are compiled away into *CHAM* rules for each resource formula to which they apply. By definition 5, the par-components of a resource formula R are the multisets of atoms obtained by recursively applying the decomposition inference figures to R till only atomic formulae remain. When there is only one par-component (second line of the table in the definition above), the decomposition is purely local to the subsolution in which it occurs. On the other hand, when there are zero or more than one par-components (first and third line, respectively, in the table), the content of the global solution is modified, as the whole subsolution where the decomposition applies may disappear or may be cloned a number of times; this global behavior is achieved via the use of the airlock mechanism of the *CHAM*, which involves a form of sequentialization.

We can now formally state the equivalence between *LO* computations and *CHAM* computations as follows:

Theorem 9. *Let $\mathcal{P}$ be an LO program (with no occurrence of the broadcast marker $\hat{}$) and let Π and Π' be two proofs based on $\mathcal{P}$. We consider on one hand (regular) proof construction sequences based on $\mathcal{P}$ and on the other hand the CHAM consisting of the set of rules*

$$\{\overline{P}\}_{P \in \mathcal{P}} \cup \{\overline{R}\}_R$$

We have

$$\Pi \Longrightarrow \Pi' \text{ if and only if } \textsc{cham}(\Pi) \longrightarrow^{\star} \textsc{cham}(\Pi')$$

4.2 Implementation of the Broadcast Mechanism

The definition of the *CHAM* rules implementing the progression inference figure is first modified to account for the broadcast markers.

Definition 10. Let $P = A_1 \parr \cdots \parr A_p \parr \with A_{p+1} \parr \cdots \with A_r \multimap R$ be a program formula, we take $\overline{P}$ to be the following *CHAM* rule:

$$A_1 \ , \ \ldots \ , \ A_p \ \longrightarrow \ R \with \{A_{p+1}, \ldots, A_r\}$$

Notice that the output part and the broadcast atoms of the input part of the formula are coupled together within molecules of a new kind, of the form $R \with \alpha$ (where R is a resource formula and α a multiset of atoms), so as to express the synchronization condition captured by the regularity requirement of proof construction.

The *CHAM* rules implementing decomposition and broadcasting are presented below, using two alternative "time stamp" mechanisms. In both cases, the subsolutions representing the living agents are indexed with a time stamp, but, in the first case, time stamps form a totally ordered linear sequence, whereas the second case makes use of a partial order with a branching structure.

Linear Time Here, the global solution contains, besides the subsolutions corresponding to the living agents, a (unique) global clock written $t!$ (where t is the current global time encoded as a non negative integer) together with blocks of messages of the form $t!\alpha$ where t is a time stamp and α a multiset of atoms. These blocks represent broadcast messages and need to be prefixed with a time stamp in order to ensure that agents read them only once. For each time stamp t lower than the current global time, there is one and only one block of messages prefixed with t, and there are no blocks of messages prefixed with time stamps larger than or equal to the current global time. The idea is that each agent indexed with a given time stamp t has already read all the messages prefixed with time stamps lower than or equal to t, and still need to read the messages prefixed with time stamps greater than t. The synchronization is achieved as follows.

- When an agent is indexed with a time stamp t lower than the current global time, and hence attached to a unique block of messages, this block of messages is read by the agent, i.e. it is added to its current state. This is achieved by the following *CHAM* rule. We take *broadcast-rule* to be the set consisting of this single rule.

  $$\{\!|\ \mathcal{S}\ |\!\}^t \ , \ t!\alpha \ \longrightarrow \ \{\!|\ \mathcal{S} \uplus \alpha\ |\!\}^{(t+1)} \ , \ t!\alpha$$

  Notice that the time stamp of the agent is incremented, so that it will never read the block of messages again. On the other hand, the block of messages itself is not discarded, as other agents may still have to read it.
- If an agent is indexed with the current global time, then, and only then, it is allowed to broadcast messages (prefixed with the global time, which then gets incremented) and decompose resource formulae.

  Definition 11. Let R be a resource formula, and α_o be a multiset of atoms, we take $\overline{R \with \alpha_o}$ to be the following *CHAM* rule:

  $$\{\!|\ R \with \alpha_o \lhd \mathcal{S}\ |\!\}^t \ , \ t! \ \longrightarrow \ t!\alpha_o \ , \ \{\!|\ \mathcal{S} \uplus \alpha_1\ |\!\}^{(t+1)} \ , \ \ldots \ , \ \{\!|\ \mathcal{S} \uplus \alpha_n\ |\!\}^{(t+1)} \ , \ (t+1)!$$

  where $\|R\| = \{\alpha_1, \ldots, \alpha_n\}$.

Notice that each application of any of these rules grabs the global clock, so that broadcast operations are here sequentialized. Furthermore, the agent(s) resulting from the decomposition of R is (are) indexed with the time stamp $t+1$, so that they cannot read the messages in α_o, which are prefixed with t. This is indeed required since, in LO, broadcast messages are immediately discarded from the state of the agent which initiates the broadcast, before the transition specified by R is performed.

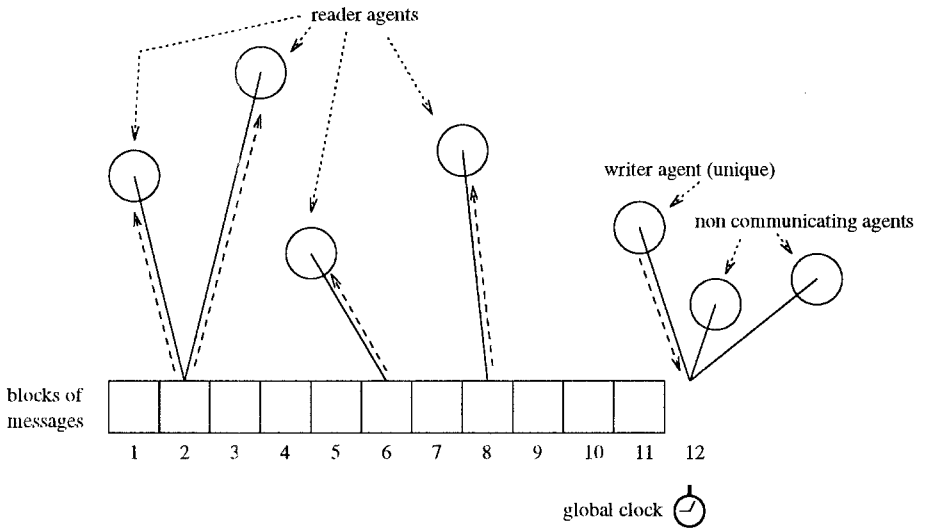

Fig. 2. The linear-time *CHAM* at work

Figure 2 shows the system of agents at work. At any time, each agent can either be a reader, a writer or non-communicating. There might be several simultaneous read operations[2], whereas only one write operation is allowed at a time. Of course, evolutions inside the agents (subsolutions) are completely independent and may occur in parallel.

[2] It could be argued that simultaneous reading is not implemented by the *CHAM* described above, because the block of messages in the *broadcast-rule* is grabbed by the reading agent. This is in fact an instance of a general problem of the *CHAM* framework: as mentioned in [CMR92] the *CHAM* cannot distinguish between the situation where some item is preserved by a rewrite rule (as intended for the block of messages in the *broadcast-rule*) and the situation where the same item is cancelled and then generated again (in which case one could argue that simultaneous reading is excluded). The CHARM framework [CMR92], offers a solution to this problem. We have left a study of using the CHARM framework instead of the *CHAM* framework for future research.

We now map *LO* proofs into *CHAM* solutions.

Definition 12. Let Π be a proof and t be a time stamp. We take $\text{CHAM}(\Pi, t)$ to be the *CHAM* solution consisting of the molecule $t!$ together with the subsolutions of the form $\{\!|\ \mathcal{C}\ |\!\}^t$ where $\mathcal{C}$ is the context at an open leaf of Π.

In other words, this mapping assumes that each open leaf of the proof is mapped into a subsolution indexed with the global time (this amounts to a global synchronization of all the agents). Furthermore, given a solution $\mathcal{S}$, we take $\text{FILTER}(\mathcal{S})$ to be the solution obtained by deleting from $\mathcal{S}$ all the blocks of messages (of the form $u \hat{} \alpha$). We can now formally state the equivalence between *LO* computations and *CHAM* computations as follows:

Theorem 13. *Let $\mathcal{P}$ be an LO program, let Π and Π' be two proofs based on $\mathcal{P}$ and let t be any time stamp. We consider on one hand (regular) proof construction sequences based on $\mathcal{P}$ and on the other hand the CHAM consisting of the set of rules*

$$\{\overline{P}\}_{P \in \mathcal{P}} \cup \{\overline{R \hat{} \alpha}\}_{R,\alpha} \cup \textit{broadcast-rule}$$

We have

$$\Pi \Longrightarrow \Pi' \text{ if and only if } \exists t', \mathcal{S} \text{ s.t. } \begin{cases} \text{CHAM}(\Pi, t) \longrightarrow^\star \mathcal{S} \\ \text{CHAM}(\Pi', t') = \text{FILTER}(\mathcal{S}) \end{cases}$$

Branching Time Here, the time stamps are finite ordered sequences of positive integers. The empty sequence is written ϵ and the expression $k.t$ denotes the sequence with first element k (an integer) and tail t (a sequence itself). The global solution contains, besides the subsolutions corresponding to the living agents, four kinds of floating molecules:

- Branching points of the form $\&_n^t$ (where $n \geq 2$) encoding a transition, at time t, with n output states (creation by cloning).
- Terminating points of the form $\top^t$ encoding a transition, at time t, with no output state (termination).
- Upward and downward blocks of messages of the form, respectively $t \uparrow \alpha$ and $t \downarrow \alpha$ where t is a time stamp and α is a multiset of atoms, implementing message propagation.

In fact, the structure of the time stamps mimics that of the proof tree: each time stamp identifies a unique node in the proof. Notice that we consider here the possibility of branching points with more than two branches, since decomposition of resource formulae is executed at once and may yield any number of par-components.

Definition 14. Let R be a resource formula, and α_o be a multiset of atoms, we take $\overline{R \hat{} \alpha_o}$ to be the following *CHAM* rule:

If	Then $R \hat{} \alpha_o$ is the rule
$\|\|R\|\| = \emptyset$	$\{\!\| R \hat{} \alpha_o \lhd \mathcal{S} \|\!\}^t \longrightarrow t \downarrow \alpha_o , \top^t$
$\|\|R\|\| = \{\alpha\}$	$\{\!\| R \hat{} \alpha_o \lhd \mathcal{S} \|\!\}^t \longrightarrow t \downarrow \alpha_o , \{\!\| \mathcal{S} \uplus \alpha \|\!\}^t$
$\|\|R\|\| = \{\alpha_1, \ldots, \alpha_n\}$ $(n \geq 2)$	$\{\!\| R \hat{} \alpha_o \lhd \mathcal{S} \|\!\}^t \longrightarrow t \downarrow \alpha_o , \&_n^t , \{\!\| \mathcal{S} \uplus \alpha_1 \|\!\}^{1.t} , \ldots , \{\!\| \mathcal{S} \uplus \alpha_n \|\!\}^{n.t}$

The propagation of the broadcast messages is achieved by the set *broadcast-rules* consisting of the following four *CHAM* rules.

$$
\begin{array}{lcl}
j.t \downarrow \alpha , \&_n^t & \longrightarrow & \{i.t \uparrow \alpha\}_{i=1\ldots n \wedge i \neq j} , t \downarrow \alpha , \&_n^t \\
t \uparrow \alpha , \&_n^t & \longrightarrow & \{i.t \uparrow \alpha\}_{i=1\ldots n} , \&_n^t \\
t \uparrow \alpha , \top^t & \longrightarrow & \top^t \\
t \uparrow \alpha , \{\!\| \mathcal{S} \|\!\}^t & \longrightarrow & \{\!\| \mathcal{S} \uplus \alpha \|\!\}^t
\end{array}
$$

The broadcast mechanism works in the following way; The "broadcaster" is always a living agent, hence an open leaf of the proof tree mapped into a sub-solution of the *CHAM*, indexed with a time stamp t identifying the position of the leaf in the proof tree. In order to broadcast a block of messages to the entire tree, the block is recursively propagated downwards in the tree, using the information of the time stamp, to all the ancestors of the agent. Furthermore each time a downward propagation step is performed at a node, upward propagation steps are triggered at that node. Of course, upward propagation is not initiated on the branch where the downward propagation takes place, but only on sibling branches. This is captured by the first rule in *broadcast-rules*. The last three rules implement in an obvious way upward propagation.

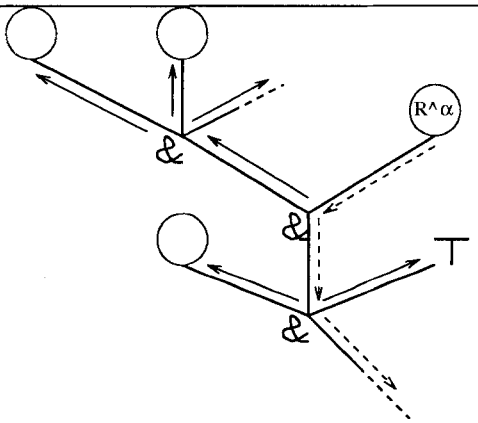

Fig. 3. The branching-time *CHAM* at work

Figure 3 illustrates the propagation mechanism. Dashed arrows represent downward propagation steps along the branch of the ancestors of the agent

requesting the broadcast. Plain arrows represent the upward propagation steps — ultimately reaching the whole tree — triggered by these ancestors. Propagation and delivery of messages may here take place simultaneously; multiple concurrent writer agents are allowed as well as multiple concurrent readers[3].

We now map *LO* proofs into *CHAM* solutions.

Definition 15. Let Π be a proof. We take $\text{CHAM}(\Pi)$ to be the *CHAM* solution consisting, for each node at position t in Π, of a molecule of the form $\&_n^t$ if the node is a branching node of Π, or a molecule of the form $\top^t$ if the node is a termination node of Π, or a subsolutions of the form $\{\!|\ \mathcal{C}\ |\!\}^t$ if the node is an open leaf labeled with the context $\mathcal{C}$.

Thus, the whole structure of the proof tree is mapped into the *CHAM* solution. We can now formally state the equivalence between *LO* computations and *CHAM* computations as follows:

Theorem 16. *Let $\mathcal{P}$ be an LO program, let Π and Π' be two proofs based on $\mathcal{P}$. We consider on one hand (regular) proof construction sequences based on $\mathcal{P}$ and on the other hand the CHAM consisting of the set of rules*

$$\{\overline{P}\}_{P\in\mathcal{P}} \cup \{\overline{R\,\hat{}\,\alpha}\}_{R,\alpha} \cup \textit{broadcast-rules}$$

We have

$$\Pi \Longrightarrow \Pi' \text{ if and only if } \text{CHAM}(\Pi) \longrightarrow^{\star} \text{CHAM}(\Pi')$$

5 Discussion

5.1 Constructing Proofs in Linear Logic

As shown in [AP91a, And92], *LO*, without the mechanism for broadcast communication, is sound and complete with respect to Linear Logic [Gir87]; in other words, (*i*) any proof in *LO* can be trivially mapped into a proof in Linear Logic, and (*ii*) any proof in Linear Logic can be transformed, by permutation of the inference figures, into a proof in normal form (called a "focusing" proof in [And92]) which can then be represented by a proof in *LO*[4]. As a matter of fact, *LO*, by trading explicit exponentials (responsible in Linear Logic for marking "reusable" formulae) with a "spatial" separation in sequents (permanent entities, i.e., program formulae, on the left of the provability symbol $\vdash$ and non-permanent entities, i.e., resource formulae, on the right) allows searching for Linear Logic proofs under a particularly efficient strategy, described in [And92] (see also Sec. 3); a spatial separation of a similar kind has also been adopted in some more recent developments of the Linear Logic enterprise [Gir91a, Gir91b].

On the other hand, the *LO* operational semantics is still given in terms of sequent inference figures, and thus maintains those aspects of artificial serialization of the computation which are inherent to sequent proofs: namely, the fact

[3] Considerations similar to those of footnote 2 apply here too

[4] In fact, in a slight extension of *LO* described in [And92] under the name LinLog

that all inference steps on a branch of proof must appear in sequence, although in principle many of them could be performed in parallel. This problem is tackled within Linear Logic itself, where the concept of "Proof Nets" has been proposed as an alternative to sequent proofs in [Gir87]: Proof Nets permit a highly parallel encoding of Linear Logic proofs. However, they are based on a well-formedness criterion (the "short trip" condition) which is rather cumbersome to handle in the process of proof construction; thus, they appear better suited for computational frameworks (e.g. Interaction Nets [Laf90]) where proofs are *normalized* (perhaps by multiple normalization steps taking place simultaneously) rather then directly searched and constructed. Our *CHAM* encoding of *LO* computations retains some aspects of desequentialization offered by Proof Nets, but is better suited to the case of proof construction.

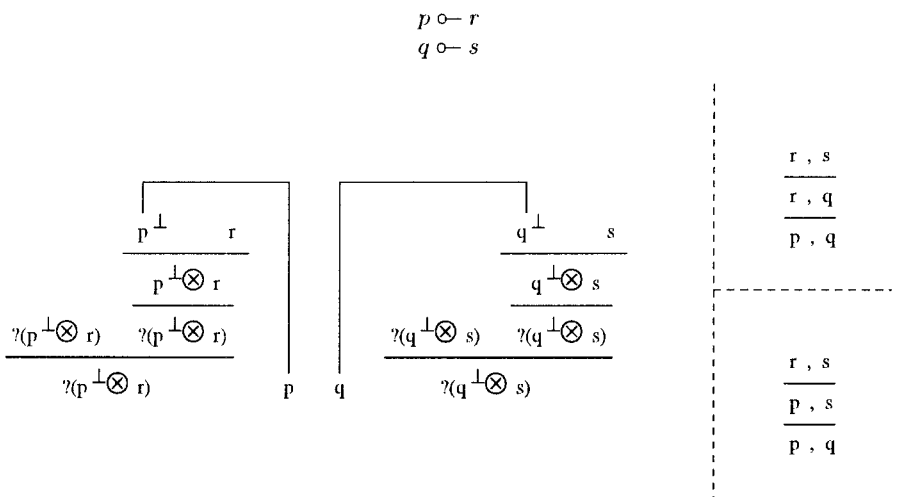

Fig. 4. A potentially simultaneous multiple transition is represented as a single proof net (left) or as two sequent proofs (right).

In particular, program formulae which are purely multiplicative (i.e. contain no occurrence of the additive connective &) can be applied simultaneously in the *CHAM* solution — provided they do not share a resource in their input part. Consider for example the program of Fig. 4. In the *CHAM* encoding, the two formulae may apply simultaneously to any solution of the form $\{\!|\, p, q, \mathcal{S} \,|\!\}$, and lead to the solution $\{\!|\, r, s, \mathcal{S} \,|\!\}$. This is captured in the proof net representation of the same transition (see Fig. 4), where the application of the two formulae[5] are performed in two distinct parts of the net and hence are not sequentialized. In

[5] The program formulae are represented by $?(p^{\perp} \otimes r)$ and $?(q^{\perp} \otimes s)$ in the Proof Net.

the sequent proof system, on the other hand, two syntactically distinct proofs are possible (see Fig. 4), which differ only inessentially in the (arbitrary) sequence of application of the formulae.

On the other hand, program formulae containing the additive & must be sequentialized. In our *CHAM* based true concurrency operational semantics, this is realized by the use of the airlock mechanism in the expression of the rules for the decomposition of resource formulae containing the additive, allowing the duplication of whole subsolutions. Similarly, proof nets introduce the concept of "box" to deal with this problem.

Clearly, one could experiment also with other frameworks related to *CHAMs* to implement our approach: suitable candidates are CHARMs [CMR92] (see footnote 2), rewriting logics [Mes92a] [6], contextual nets [MR91] etc.

The proof construction mechanism called "instantiation" in section 3.3, can be shown sound and complete with respect to Linear Logic, as shown in [AP91a], via the Phase Semantics, introduced in [Gir87] to provide a model-theoretic (tarskjian) characterization of Linear Logic. Basically, soundness and completeness rely on the idea that generating contexts through proof construction amounts to enumerating phases in the phase space corresponding to the denotation of the given program. In this way, the "copying" operational meaning of the & connective is exploited not just for duplicating preexisting information, but also for propagating new information. However, in order to make this approach practical, there is need of the broadcast marker ^ to give the user control on which phases are effectively generated; as a consequence, this creates a form of incompleteness. Still, the "focusing" strategy for searching proofs can be maintained, by intertwining it with synchronization mechanisms to handle communication; this is directly reflected in the fact that the *CHAM* encodings for *LO* with broadcasting are simple extensions of the non-broadcasting case, where the extensions specifically concern the delivery and the reception of information.

5.2 Broadcasting

We have seen that the main challenges about implementing broadcasting in *LO* are in keeping track of the dynamic reconfigurability of the system (creation and deletion of agents at run-time), and ensuring soundness in the delivery of the messages. This kind of challenges is in general to be met in adding broadcasting to systems based on one-to-one communication. We have proposed two solutions which both make use of a notion of time to overcome such difficulties; thus, messages are time-stamped, and agents themselves have an internal time through which it can be checked whether a given message must or must not be delivered to a given agent. The "linear time" solution relies on the idea that agents regulate their own clocks with respect to a global clock, through which messages are also time-stamped; agents can only read messages which are not in their present or in their past; the internal clock of an agent advances each time

[6] As a matter of fact, a broadcast mechanism has been recently implemented on top of the rewriting logic programming language MAUDE [Mes92b].

an agent reads a message or writes one, in which case the global clock gets itself updated through the action of writing. The "branching time" solution relies on the idea of time-stamping agents at the time of their creation; sibling agents all get different stamps, initiating alternative futures evolving from a common past; messages are themselves time-stamped with the stamp of their creator, and cannot be read by any agent who belongs to the same present or the same future of the broadcast message. In this case, there is no global clock, but only local times, starting from the time of the oldest agent, which is stamped with ϵ. Both solutions have their advantages and their trade-offs: linear time is very simple, and entails a straightforward way of implementing parallel reading, but is completely sequential in the writing, as each writer agent must first take possession of the global clock and release it after the write is performed; branching time permits more parallelism in the writing, as there is no notion of a unique global clock, but is more complicated from the point of view of reading, as information must flow through a sideway-downward/upward propagation scheme along the branches of the proof tree. We can make conjectures about which machine architectures are more appropriate for each solution: shared memory architectures appear particularly well-suited for linear time, as the time stamps could be seen as different locations in the memory where the messages are stored and agents actively go and fetch them; highly distributed, message passing architectures fit well instead with branching time, as open nodes in the proof (agents) can be identified with nodes in a network and their ancestors with communication points handling the routing information.

The type of broadcasting in *LO* programs is closely related to the model studied in [Pra91]. These studies are concerned with the externally observable behavior of processes communicating via broadcasting, thus assuming broadcasting as a primitive mode of communication. However, it is seldom (if ever) the case that parallel or distributed implementation platforms provide broadcast communication as a primitive. Instead one-to-one asynchronous message passing seems to be predominant, and we have therefore focused on how to implement (in an abstract sense) broadcast communication when one-to-one asynchronous message passing is assumed. Recently in the (theoretical) object oriented community the asynchronous one-to-one message passing paradigm has been studied in its own right [HT91, Nie91].

Acknowledgement

Acknowledgements are due to the anonymous Tapsoft referees for their insightful comments.

References

[ACP92] 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*. MIT Press, 1992. To appear.

[And92] J-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.

[AP91a] J-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Proc. of OOPSLA'91*, Phoenix, Az, U.S.A., 1991.

[AP91b] J-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3+4):445–473, 1991.

[APB91] J-M. Andreoli, R. Pareschi, and M. Bourgois. Dynamic programming as multi-agent programming. In *Proc. of the OOPSLA'90/ECOOP'91 workshop on Object-based concurrent computing*, Lecture Notes in Computer Science (612), Genève, Switzerland, 1991. Springer Verlag.

[BAP92] M. Bourgois, J-M. Andreoli, and R. Pareschi. Extending objects with rules, composition and concurrency: the *lo* experience. In *Proc. of the OOPSLA'92 workshop on Object-Oriented Programming Languages — The Next Generation*, Vancouver, Canada, 1992.

[BB90] G. Berry and G. Boudol. The chemical abstract machine. In *Proc. of the 17th ACM Symposium on Principles of Programming Languages*, San Francisco, Ca, U.S.A., 1990.

[BCLM88] J-P. Banâtre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4(2):133–145, 1988.

[CMR92] A. Corradini, U. Montanari, and F. Rossi. Concurrency and hiding in an abstract rewrite machine. In *Proc. of the International Conference on 5th Generation Computer Systems*, Tokyo, Japan, 1992.

[Gir87] J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir91a] J-Y. Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, 1(3), 1991.

[Gir91b] J-Y. Girard. On the unity of logic, 1991. Preprint, Université de Paris 7.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP'91*, Genève, Switzerland, 1991.

[Laf90] Y. Lafont. Interaction nets. In *Proc. of 17th ACM Symposium on Principles of Programming Languages*, San Francisco, Ca, U.S.A., 1990.

[Mes92a] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 93:73–155, 1992.

[Mes92b] 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, 1992. To appear.

[MR91] U. Montanari and F. Rossi. Contextual nets. Technical report, Università di Pisa, Pisa, Italy, 1991.

[Nie91] O. Nierstrasz. Towards an object calculus. In *Proc. of the OOPSLA'90/ECOOP'91 workshop on Object-based concurrent computing*, Lecture Notes in Computer Science (612), Genève, Switzerland, 1991. Springer Verlag.

[Pra91] K.V.S. Prasad. A calculus of broadcasting systems. In *Proc. of the 16th Colloquium on Trees in Algebra and Programming*, 1991.

[Sar89] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Pittsburg, Pa, U.S.A., 1989.