International Journal of Parallel Programming, Vol. 16, No. 6, 1987

Partial Ordering Models for Concurrency Can Be Defined Operationally

Pierpaolo Degano[1] and Sergio Marchetti[2]

Received December 1987; revised June 1988

Labelled rewriting systems are shown to be powerful enough for defining the semantics of concurrent systems in terms of partial orderings of events, even in the presence of non standard operators like N that is not expressible by means of concurrency and sequentialization. This contrasts with Pratt's claim.(1) The main operators proposed by Pratt are used here to construct terms denoting concurrent systems, the behavior of which consists of partially ordered multisets defined operationally.(2) Fully abstractness of the denotational semantics as defined in Ref. 1 with respect to the operational one is finally proved.

KEY WORDS: Concurrency; partial orderings; pomsets; labelled rewriting systems; operational semantics; denotational semantics; fully abstractness.

1. INTRODUCTION

Many models have been proposed in the literature for describing distributed concurrent systems considered as sets of sequential processes which cooperate in accomplishing a task. These sequential processes may be possibly located in different places, and each of them performs a specific sub-task, at its own processing speed, with its own local clock, either in an independent manner or through synchronizations with other processes for communicating intermediate results. These models have been historically developed following two main lines.

[1] Dipartimento di Informatica—Università di Pisa, Corso Italia 40, I-56100 PISA, Italy, e_mail: degano@dipisa.uucp.

[2] Selenia S.p.A., Via Tiburtina Km 12.4, I-00100 ROMA, Italy, presently at LIST S.p.A., Piazza Mazzini 6, I-56100 PISA, Italy.

0885-7458/87/1200-0451$05.00/0

In the first approach, often referred to as the interleaving approach, e.g., see Refs. 3–10, features describing parallel composition and nondeterminism are added to sequential languages or to models for them (see, e.g., CSP[5] or CCS[9]). In this framework, a concurrent system is represented by a term E, and its operational semantics is given through labelled transition systems. More precisely, a transition $E - a \rightarrow E'$ models the fact that E evolves, by performing an event observed as a, to another concurrent system E'. The state of a concurrent system is then represented as a monolithic entity, and thus a global time and a centralized control are implicitly assumed. Consequently, a total ordering among possibly spatially separated and causally independent events is imposed, and concurrency is expressed by the fact that concurrent events can occur in any order. In other words, the operator of parallel composition is not primitive since it is reduced to nondeterminism and interleaving. A major advantage of using transition systems is that they can be defined in the so-called Structured Operational Semantics style[11] (SOS for short), via axioms and inference rules. Following this style, a transition for a term is deduced by inducing on its syntactic structure in a merely compositional way.

The second main line followed in describing distributed concurrent systems is often referred to as the true concurrent, or the partial ordering approach.[1,12–21,28] Petri Nets[19] are perhaps the best known model within this framework. Their starting point are nondeterministic automata which have been enriched by giving states an additional structure of set to represent distributed states, and by allowing transitions to involve only some of the processes present in the actual state. Thus, neither a global state nor a global clock are assumed. The behavior of systems is represented through the causal relations among the events performed by the components of their distributed state.[22] The resulting abstract machine is then much more complex than the one based on labelled transition systems, thence the conceptual simplicity of the interleaving framework is lost in this approach. However, we stand firmly on the partial ordering side, even though its theory is not completely satisfactory, because it offers a definitely better, always closer, and often simpler description of reality. See Pratt's[1] detailed discussion about this issue. As an example of the lack of expressive power of the interleaving, see Fig. 1a) where there is an instance of the well known N-structure which is not expressible in terms of concurrency and sequentialization.

Pratt[1] in a recent paper, strongly advocates the use of partially ordered multisets, called *pomsets*, in modelling concurrency, further supporting an increasingly growing interest in the true concurrent approach to the semantics of concurrent systems. In his paper Pratt shows some operators for defining a denotational model, and gives no operational

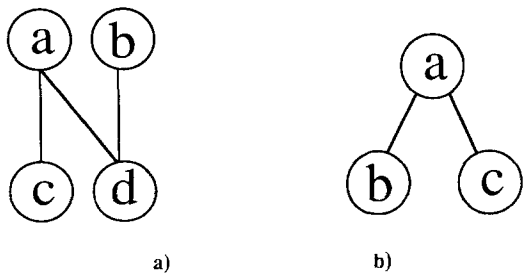

Fig. 1. Two pomsets. Events are represented as labelled circles, the partial ordering through its Hasse diagram growing downwards.

semantics to them, since he complains about defining operationally "the meaning of expressions [...] by reductions between expression" because this "forces an interleaving view of concurrent computation."[1]

We have been challenged by this complaint to provide (parts of) his model with a distributed, truly concurrent operational semantics. This is because we are strongly convinced that any programming language should be provided with a formal operational semantics coupled with a denotational one, let alone with an axiomatic semantics, and that these semantics must be proved equivalent. While the denotational definition is particularly suited for reasoning about programs, the operational one gives a precise yet intuitive description of the language, as long as the chosen abstract machine is. Moreover, the denotational semantics does not provide us with any hint about implementation issues, while the operational one gives firm guidelines and points out difficulties and suggests solutions which can be more easily devised in its relatively abstract setting. When designing a concurrent language, which is still a hard research task, it is even more important to compare its denotational and operational semantics in order to remove any inadequacies, ambiguities, inconsistencies, and in order to properly monitor its behavior.

Our starting point has been the previous work carried out by Ugo Montanari and by the first author,[15,16] which aim at defining a setting in which concurrent languages could be equipped with partial ordering semantics, both operational and denotational, and in which compositionality of the interleaving models could be combined with the expressivity of the true concurrent one. Also Refs. 2, 23–25 are relevant to this issue in that they provide CCS and CSP with a concurrent and distributed semantics based on partial orderings. In order to show that the proposed technique is powerful enough, we will consider some relevant operators defined in Ref. 1 for composing pomsets. More precisely, we will deal with sequentialization (denoted by ";"), parallel composition ("||"),

iteration of parallel composition ("†") and **N** operator. We will not consider other operators, for the sake of brevity and because they have either already been studied (e.g., nondeterminism[2,24,25]) or they are similar to the ones we work with (e.g., iteration of sequentialization, "*"). Note that we will not deal with communication, since Pratt[1] defines no semantic operator for it, although an operational treatment of synchronization and communication is straightforward.[2,24,25] The most difficult task we found was defining rules for expressing the behavior of **N**-terms which have never been operationally dealt with before, or explicitly discarded.[12] Furthermore, we stress the importance of the **N**-structure, since it is the typical partial ordering not definable using series/parallel operators, only.

Our operational semantics requires a few steps to be defined. We still borrow from the interleaving approach the representation of (the states of) concurrent systems as terms, and the SOS style of defining their evolution, thus guaranteeing that our operational semantics is compositional. Our goal is to represent the evolution of a system as the causal relations among the events performed by sub-parts of its state. Hence, we will first decompose the term denoting a state into its *sequential processes*, namely into those sub-terms which may perform actions independently of each other. For instance, from the term $b \| c$ we obtain the following two sequential processes $b|\mathrm{id}$ and $\mathrm{id}|c$. Tag "|id" records that sub-term b was in the left context of a parallel composition, and that it was enabled to perform an action b on its own; symmetrically for the other sequential process. The set of sequential processes obtained from a term in this way will then represent the *distributed state* of a system.

The dynamics of a system is then described by a set of *rewriting rules* defined in the SOS style, via axioms and inference rules. A rewriting rule specifies how only some of the sequential processes in a distributed state may evolve, leaving the remaining ones idle. As usual, these rules are applied to a distributed state to get *computations*, which will finally be *observed* as pomsets.

More in detail, a rewriting rule has the form $I - [a, \mathscr{R}] \rightarrow p$, where I represents a set of sequential processes which may evolve by performing action a to the sequential process p. Thus, we may say that the sequential processes in I *cause* p through a. The relation $\mathscr{R}$ over sequential processes which also labels a rewriting rule gives additional information about the causal relation, since it may happen that there are other sequential processes (forming a state J) which are caused by (some) sequential processes in I (forming a state $I' \subseteq I$), but not by a. Since causality will later on be represented by a partial ordering relation $\leqslant$, we will express this fact as $\{p' \leqslant p \mid p' \in I' \text{ and } p \in J\}$, or $I \leqslant J$, for short. The intended meaning of applying a rewriting rule to a distributed state is that the set I

occurring in it can be replaced, after showing an event (labelled by) a, by sequential process p and all the sequential processes in I'. In this way we obtain the new distributed state. As an example, consider the system state denoted by term $a;(b \| c)$ which has only one sequential process. After performing event a, state $1;(b \| c)$ is reached (term 1 represents the process which cannot perform any event) which again has a single sequential process. Nevertheless, the parallel composition of b and c is enabled; for instance sequential process $b|\mathrm{id}$ can evolve by performing b to the sequential process $1|\mathrm{id}$, independently of the other sequential process $\mathrm{id}|c$. A rewriting rule will be deducible recording both the evolution of $b|\mathrm{id}$ and, through relation $\mathscr{R}$, that $1;(b \| c)$, *but not* event b, causes $\mathrm{id}|c$; symmetrically when the sequential process $\mathrm{id}|c$ moves first, in temporal ordering. In other words, relation $\mathscr{R}$ expresses the fact that $\mathrm{id}|c$ may perform event c concurrently with event b. Formally, we have in the first case the rewriting rule $\{1;(b \| c)\} - [b, 1;(b \| c) \leqslant \mathrm{id}|c] \to 1|\mathrm{id}$, pictorially represented in Fig. 2; in the second case we will have rewriting rule $\{1;(b \| c)\} - [c, 1;(b \| c) \leqslant b|\mathrm{id}] \to \mathrm{id}|1$.

A *computation* is a sequence of sets of sequential processes, representing distributed states of systems, and of rewriting rules, representing the evolution of system sub-parts. From a computation, we finally obtain the wanted pomset, by keeping the essence of the causal relations contained in the rewriting rules, in spite of their strictly sequential application. Of course, in this example we may have two computations, the first one when event b occurs before c and the other with the inverse temporal ordering. In both cases, we get the same pomset, depicted in Fig. 1b.

Finally, the *operational semantics* of a term will be the set of all pomsets obtained in this way. A richer structure can also be given to this set,

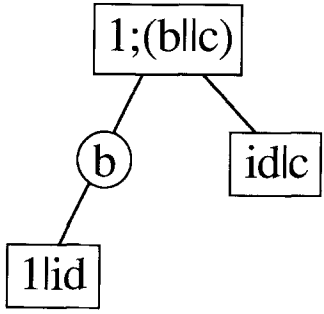

Fig. 2. A graphical representation of rewriting rule $\{1;(b \| c)\} - [b, 1;(b \| c) \leqslant \mathrm{id}|c] \to 1|\mathrm{id}$. Sequential processes are represented as labelled boxes.

e.g., by defining event structures[21] or NMS's,[16] but we have chosen a simpler setting to easily compare Pratt's approach with ours. Indeed, we will define, for the common sub-language, the *denotational semantics* of a term by using the operators on pomsets introduced by Pratt[1]; then, we will prove that the denotational and the operational semantics coincide in quite a strong sense, since the former proves to be fully abstract with respect to the latter. Due to the compositionality of both semantics, the proofs of these results are carried out by possibly boring, but straightforward structural induction.

In literature,[12,13] there are other approaches aiming at associating partial orderings to terms. However, the proposed techniques are not fully operational, unlike ours. Actually, these papers use transition systems, rather than rewriting systems. Thus, there is no notion of distributed state, and moreover their transitions directly carry partial ordering as labels, thus losing the notions of elementary step and of computation, as concatenation of them, in favor of a more denotational style.

The paper is organized as follows. Section 2 introduces the simple concurrent language we deal with, and the set of rewriting rules which specify the elementary steps of computations. In the same section we also extract pomsets out of computations, while in Section 3 we define the denotational semantics, we prove its equivalence with the operational one, and the fully abstractness of the former with respect to the latter.

2. SYNTAX AND OPERATIONAL SEMANTICS

In this section we introduce an abstract syntax for the simple concurrent language we are considering, and the set of rewriting rules that specify its operational semantics.

A subset of the combinatorial operators proposed by Pratt[1] are dealt with here. We will not consider all operators, for the sake of brevity and because some of them have either already been studied (e.g., nondeterminism) or they are similar to the ones we work with (e.g., iteration of sequentialization, "*"). Extending the definitions given in this section to deal with such operators is only a clerical matter. The definition of the syntax of the language follows.

Definition 2.1. Given a finite alphabet Σ of *actions*, ranged over by a, b,..., an *agent* is a term defined by the following BNF-like grammar.

$$E ::= 1 \mid a \mid E; E/E \| E/E^{\dagger} \mid \mathbf{N}(E, E, E, E)$$

The intuition behind agents is as follows. Agent 1 represents the empty pomset. Operationally, 1 is an agent which cannot perform any actions.

Agent a denotes the pomset with only one event labelled by a, *i.e.*, the agent which can perform action a and then stop. Agent $E'; E''$ stands for the concatenation of the pomsets of E' and E'', that can be generated by building on these pomsets in sequence. Agent $E' \| E''$ denotes the disjoint union of the pomsets of E' and E'', obtained operationally by independently generating the events of the two operands of "$\|$". Thus, neither communication nor even synchronization are allowed. Agent $E^\dagger$ represents the set of finite pomsets denoted by agents $1, E, E \| E, E \| E \| E, \ldots$. Again, one of the pomsets of $E^\dagger$ is operationally determined by independently generating the events of every copy of E. Finally, the agent $\mathbf{N}(E_1, E_2, E_3, E_4)$, denoting the **N**-shaped pomset, is operationally dealt with as $(E_1; E_3) \| (E_2; E_4)$, with the additional constraint that all the events of E_1 precede those of E_4.

Our goal is to define the causal relations among the events performed by sub-parts of a distributed state of a system. Thus, we have to single out of an agent those subterms which represent its *sequential processes*, which are, roughly speaking, all those sub-terms with a sequential operator at top-level. First, we give the syntax for them.

Definition 2.2. (Sequential processes) A *sequential process*, or *process* for short, is a term defined by the following BNF-like grammar.

$$P ::= 1 \mid a \mid E^\dagger \mid P; E \mid P | \mathrm{id} \mid \mathrm{id} | P \mid \langle P, E \rangle | \mathbf{N} | \mathbf{N} | \langle P, E \rangle \mid 1 | \mathbf{N},$$

where E is an agent.
We will use p and I, J (possibly indexed) to range over sequential processes and sets of sequential processes, respectively.

Intuitively speaking, a sequential process of an agent E represents one of its sub-agents, together with its access path, used to take into account in which context within E the sequential process p was plugged. As an example, $a | \mathrm{id}; (c \| d)$ and $\mathrm{id} | b; (c \| d)$ are the sequential processes of the agent $a \| b; (c \| d)$. The access path of the former is $| \mathrm{id}; (c \| d)$ which records, via "$| \mathrm{id}$", that the sequential process was put in parallel with another one of the form $\mathrm{id} | p$, and also records, via "$;(c \| d)$", that it was followed by agent $(c \| d)$. Of course, we consider agents 1 and a as sequential. We consider also $E^\dagger$ as sequential, for the specific technique we will introduce in Definition 2.5 to generate the needed copies of E. Recursively, tags are attached to a sequential process in order to reflect the syntactic structure of the agent from which it is originated. More in detail, tag "$;E$" may be attached to a sequential process p obtaining the sequential process $p; E$ which expresses the fact that $p; E$ was part of an agent of the form $E'; E$. We also call E the *continuation* of p. As already seen in the previous example, both sequential processes have as continuation the agent $c \| d$. In the

case of parallel composition, we replace $\|$ by two unary tags, "|id" and "id|", which record that there are sequential processes that can evolve concurrently. Back to our example, the sequential process $a|\mathrm{id}; (c \| d)$ is enabled to perform action a concurrently with action b which can be performed by $\mathrm{id}|b; (c \| d)$. Analogously, tags "$|\mathbf{N}$" and "$\mathbf{N}|$" represent both the context $\mathbf{N}(E_1, E_2, E_3, E_4)$ and the fact that E_1 and E_2 are enabled to proceed independently. Information about the remaining two operands E_3, E_4 is kept in the second element of the pairs. Finally, also $1|\mathbf{N}$ will be considered for technical reasons as a sequential process: it will be used in actual computations as a sentinel keeping track that all the sequential processes of E_1 are terminated.

Sequential processes are obtained from agents via the following decomposition function.

Definition 2.3. (Decomposition function) Function dec, defined inductively, maps agents to sets of sequential processes.

$$\mathrm{dec}(1) = \{1\} \qquad \mathrm{dec}(a) = \{a\}$$

$$\mathrm{dec}(E^\dagger) = \{E^\dagger\} \qquad \mathrm{dec}(E_1; E_2) = \mathrm{dec}(E_1); E_2$$

$$\mathrm{dec}(E_1 \| E_2) = \mathrm{dec}(E_1)|\mathrm{id} \cup \mathrm{id}|\mathrm{dec}(E_2)$$

$$\mathrm{dec}(\mathbf{N}(E_1, E_2, E_3, E_4)) = \langle \mathrm{dec}(E_1), E_3 \rangle |\mathbf{N} \cup \mathbf{N}| \langle \mathrm{dec}(E_2), E_4 \rangle$$

In this definition, and from now onwards, the application of a syntactic constructor to a set of sequential processes I, is understood to operate elementwise, e.g., $\langle I, E \rangle |\mathbf{N} = \{\langle p, E \rangle |\mathbf{N} \mid p \in I\}$.

Later on, we will use sets of sequential processes to represent the states which a concurrent system can pass through during a computation. We call these states *distributed*, since their components can be allocated in different places, and can proceed on their own, as we will show in a while, without requiring any centralized control. As expected, we will use $\mathrm{dec}(E)$ as the initial distributed state of a computation generating the set of pomsets denoted by E. Function dec is obviously injective, but not all the distributed states reachable in a computation are in direct correspondence with agents via dec. The main reason is that a distributed state could contain sequential process $1|\mathbf{N}$ which does not occur in the decomposition of any agent (recall that it represents termination of all the sequential processes of the first agent of an N-agent). An instance of such a distributed state is $\{p_6, p_7, p_3\}$ of computation ξ_1 in Example 2.3. As a matter of fact, it is possible to define a many-to-one correspondence between sets of sequential processes and agents, at the cost of having a more intricate decomposition, relying on a relation instead of a function. For something in this line, refer to Ref. 24.

Example 2.1.

$$\mathrm{dec}((a \| b); (c \| d)) = \mathrm{dec}((a \| b)); (c \| d) = \{a | \mathrm{id}; (c \| d), \mathrm{id} | b; (c \| d)\}$$

$$\begin{aligned}
&\mathrm{dec}(\mathbf{N}(a, b, c, d) \| \mathbf{N}(e, f, g, h)) \\
&\quad = \mathrm{dec}(\mathbf{N}(a, b, c, d)) | \mathrm{id} \cup \mathrm{id} | \mathrm{dec}(\mathbf{N}(e, f, g, h)) \\
&\quad = \{\langle a, c\rangle | \mathbf{N}, \mathbf{N} | \langle b, d\rangle\} | \mathrm{id} \cup \mathrm{id} | \{\langle e, g\rangle | \mathbf{N}, \mathbf{N} | \langle f, h\rangle\} \\
&\quad = \{(\langle a, c\rangle | \mathbf{N}) | \mathrm{id}, (\mathbf{N} | \langle b, d\rangle) | \mathrm{id}, \mathrm{id} | (\langle e, g\rangle | \mathbf{N}), \mathrm{id} | (\mathbf{N} | \langle f, h\rangle)\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{dec}(\mathbf{N}(a \| b, c, d, e)) &= \langle \mathrm{dec}(a \| b), d\rangle | \mathbf{N} \cup \{\mathbf{N} | \langle c, e\rangle\} \\
&= \langle \{a | \mathrm{id}, \mathrm{id} | b\}, d\rangle | \mathbf{N} \cup \{\mathbf{N} | \langle c, e\rangle\} \\
&= \{\langle a | \mathrm{id}, d\rangle | \mathbf{N}, \langle \mathrm{id} | b, d\rangle | \mathbf{N}, \mathbf{N} | \langle c, e\rangle\}
\end{aligned}$$

Before introducing the dynamics of our formalism, we need some notation used to describe the causal relation between sets of sequential processes.

2.1. Notation

Let $\mathscr{R}$ be a binary relation over sequential processes, by $\mathscr{R} \downarrow 2$ we understand the set $\{y \mid \exists x \langle x, y\rangle \in \mathscr{R}\}$; a pair $\langle x, y\rangle$ belonging to $\mathscr{R}$, will be also written as $x \leqslant y$; given two sets of sequential processes I and J, $I \leqslant J$ will stand for $\{x \leqslant y \mid x \in I \text{ and } y \in J\}$.

Furthermore, we consider tags to be extended on $\mathscr{R}$ too, e.g.,

$$\mathscr{R} | \mathrm{id} = \{\langle x | \mathrm{id}, y | \mathrm{id}\rangle \mid \langle x, y\rangle \in \mathscr{R}\}$$

and

$$\langle \mathscr{R}, E\rangle | \mathbf{N} = \{\langle \langle x, E\rangle | \mathbf{N}, \langle y, E\rangle | \mathbf{N}\rangle \mid \langle x, y\rangle \in \mathscr{R}\}.$$

The set of rewriting rules specifying the behavior of sequential processes is defined via axioms and inference rules in the SOS style. A rewriting rule has the form $I - [a, \mathscr{R}] \to p$, the intuitive meaning of which is that the set of sequential processes I may become the process p after performing action a. Thus, we may say that each sequential process of I *causes* process p through a. The information about other sequential processes which can be caused by I but not by a is recorded in $\mathscr{R}$. The pair $\langle p_1, p_2\rangle \in \mathscr{R}$ will be written also as $p_1 \leqslant p_2$, since the same symbol $\leqslant$ will be used later to denote the causal relation of a pomset. Note that if $p_1 \leqslant p_2$, we have that $p_1 \in I$, $p_2 \neq p$ and that p_1, but *not* action a, causes process p_2. Thus, $\mathscr{R}$ records that there are agents that may perform actions which are

concurrent with a. When the rewriting rule $I - [a, \mathscr{R}] \rightarrow p$ is applied to a distributed state, it replaces the sequential processes of I with p and those of $\mathscr{R} \downarrow 2$, while showing a. Note in passing that the absence of an explicit synchronization mechanism makes set I consist of a single sequential process, except when hidden synchronizations take place, via a join of terminated processes originated by a parallel or **N** composition. For instance, look at the case when both $a|\mathrm{id}$ and $\mathrm{id}|b$ are terminated in $(a \| b); (c \| d)$, as shown in the following example. In order to make examples more readable, we will draw rewriting rules as we have already done in the Introduction: processes (events) are represented as labelled boxes (circles) and the causal relation through its Hasse diagram growing downwards. In Fig. 3 we see derivation

$$\{(1|\mathrm{id}); (c \| d), (\mathrm{id}|1); (c \| d)\}$$

$$- [c, \{(1|\mathrm{id}); (c \| d), (\mathrm{id}|1); (c \| d)\} \leqslant \{\mathrm{id}|d\}] \rightarrow 1|\mathrm{id}.$$

Its starting distributed state $\{(1|\mathrm{id}); (c \| d), (\mathrm{id}|1); (c \| d)\}$ can be reached, for instance, from agent $(a \| b); (c \| d)$ (its decomposition is in Example 2.1), by performing both concurrent actions a and b. (See also Examples 2.3.iii and 2.4.iii.)

We can now give the dynamics of our formalism: first a predicate *end* is introduced to individuate those sets of sequential processes which are terminated and will eventually be joined since they have been originated by the same (sub-)term; then rewriting rules are defined through axioms and inference rules à la Plotkin.[11]

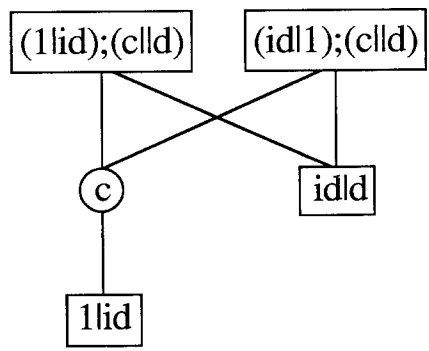

Fig. 3. A graphical representation of rewriting rule $\{(1|\mathrm{id}); (c \| d), (\mathrm{id}|1); (c \| d)\} - [c, \{(1|\mathrm{id}); (c \| d), (\mathrm{id}|1); (c \| d)\} \leqslant \{\mathrm{id}|d\}] \rightarrow 1|\mathrm{id}$.

Definition 2.4. (Terminated sets of processes) The predicate *end* is defined inductively on sets of sequential processes by the following axioms and inference rules

$\mathrm{end}(\{I\})$

$\mathrm{end}(\{E^\dagger\})$

$\mathrm{end}(I_1)$ **and** $\mathrm{end}(I_2)$ **imply** $\mathrm{end}(I_1 \,|\mathrm{id} \cup \mathrm{id}|\, I_2)$

$\mathrm{end}(I_1)$ **and** $\mathrm{end}(I_2)$ **and** $\mathrm{end}(\mathrm{dec}(E_3))$ **and** $\mathrm{end}(\mathrm{dec}(E_4))$
imply $\mathrm{end}(\langle I_1, E_3\rangle|\mathbf{N} \cup \mathbf{N}|\langle I_2, E_4\rangle)$

$\mathrm{end}(I_2)$ **and** $\mathrm{end}(I_3)$ **and** $\mathrm{end}(\mathrm{dec}(E_4))$
imply $\mathrm{end}(I_3|\mathrm{id} \cup \{I|\mathbf{N}\} \cup \mathbf{N}|\langle I_2, E_4\rangle)$

Of course, the singleton consisting of agent I is terminated. Also the set containing $E^\dagger$ must be considered as terminated, because also the empty pomset is one of the behavior of $E^\dagger$. The termination of a parallel composition, or of a **N** composition is that of all their arguments. The last rule is somehow *ad hoc*, conceived for dynamically dealing with termination of $\mathbf{N}(E_1, E_2, E_3, E_4)$ and will be better understood afterwards. We only note that sequential process $I|\mathbf{N}$ and set $I_3|\mathrm{id}$ represent what remains after the complete evolution of E_1 and E_3, respectively.

From now onwards, given two sets of sequential processes I and J, we denote their difference by I-J which is defined only if $J \subseteq I$.

Definition 2.5. (Rewriting rules) The *rewriting derivation relation* $I - [a, \mathscr{R}] \to p$ is defined as the least relation satisfying the following set of axiom and inference rules

Act)	$\{a\} - [a, \varnothing] \to I$		
Seq)	$I - [a, \mathscr{R}] \to p$	**implies**	$I; E - [a, \mathscr{R}; E] \to p; E$
	$(\mathrm{dec}(E)\text{-}I') - [a, \mathscr{R}] \to p$	**and** $\mathrm{end}(I)$	
		imply	$I; E - [a, I; E \leqslant \mathscr{R}\downarrow 2 \cup I'] \to p$
Conc)	$I - [a, \mathscr{R}] \to p$	**implies**	$I\|\mathrm{id} - [a, \mathscr{R}\|\mathrm{id}] \to p\|\mathrm{id}$
		and	$\mathrm{id}\|I - [a, \mathrm{id}\|\mathscr{R}] \to \mathrm{id}\|p$
Dagger)	$(\mathrm{dec}(E)\text{-}I') - [a, \mathscr{R}] \to p$	**implies**	$\{E^\dagger\} - [a, \{E^\dagger\} \leqslant I' \cup \mathscr{R}\downarrow 2] \to p$
	$\{E^\dagger\} - [a, \mathscr{R}] \to p$	**implies**	$\{E^\dagger\} - [a, \{E^\dagger\} \leqslant \mathrm{id}\|\mathrm{dec}(E) \cup (\mathscr{R}\downarrow 2)\|\mathrm{id}] \to p\|\mathrm{id}$
		and	$\{E^\dagger\} - [a, \{E^\dagger\} \leqslant \mathrm{dec}(E)\|\mathrm{id} \cup \mathrm{id}\|(\mathscr{R}\downarrow 2)] \to \mathrm{id}\|p$
N)	$I - [a, \mathscr{R}] \to p$	**implies**	$\langle I, E\rangle|\mathbf{N} - [a, \langle \mathscr{R}, E\rangle|\mathbf{N}] \to \langle p, E\rangle|\mathbf{N}$
		and	$\mathbf{N}|\langle I, E\rangle - [a, \mathbf{N}|\langle \mathscr{R}, E\rangle] \to \mathbf{N}|\langle p, E\rangle$
	$(\mathrm{dec}(E_3)\text{-}I') - [a, \mathscr{R}] \to p$	**and** $\mathrm{end}(I)$	
		imply	$\langle I, E_3\rangle\,|\mathbf{N} - [a, \mathscr{R}'] \to p\,|\mathrm{id}$
		where	$\mathscr{R}' = \langle I, E_3\rangle|\mathbf{N} \leqslant (\{I|\mathbf{N}\} \cup I'|\mathrm{id} \cup (\mathscr{R}\downarrow 2)\,|\mathrm{id})$

$$(\mathrm{dec}(E_4)\text{-}I') - [a, \mathscr{R}] \to p \textbf{ and } \mathrm{end}(I)$$
$$\textbf{imply} \quad \{I|\mathbf{N}\} \cup \mathbf{N}|\langle I, E_4\rangle - [a, \mathscr{R}'] \to \mathrm{id}|\, p$$
$$\textbf{where} \quad \mathscr{R}' = (\{I|\mathbf{N}\} \cup \mathbf{N}|\langle I, E_4\rangle) \leqslant (\mathrm{id}|\, I' \cup \mathrm{id}|(\mathscr{R} \downarrow 2))$$

$$(\mathrm{dec}(E_4)\text{-}I') - [a, \mathscr{R}] \to p \textbf{ and } \mathrm{end}(I_1) \textbf{ and } \mathrm{end}(I_2)$$
$$\textbf{imply} \quad \langle I_1, E_3\rangle|\mathbf{N} \cup \mathbf{N}|\langle I_2, E_4\rangle - [a, \mathscr{R}_1 \cup \mathscr{R}_2] \to \mathrm{id}|\, p$$
$$\textbf{where} \quad \mathscr{R}_1 = (\langle I_1, E_3\rangle|\mathbf{N} \cup \mathbf{N}|\langle I_2, E_4\rangle) \leqslant (\mathrm{id}|\, I' \cup \mathrm{id}|(\mathscr{R} \downarrow 2))$$
$$\mathscr{R}_2 = \langle I_1, E_3\rangle\, |\mathbf{N} \leqslant \mathrm{dec}(E_3)\, |\mathrm{id}$$

We can now comment on our axiom and rules. There is no rule for I, so this term will have only an empty behavior. Axiom *Act*) states that sequential process a can perform action a and then stop. First rule *Seq*) simply says that a set of sequential processes I with continuation E can do the same actions as I, provided that relation $\mathscr{R}$ and the caused process p are accordingly modified. The second rule is more complicated, since it has to forbid the component E of an agent $E_1; E$ to proceed until E_1 is over, and to properly set the causal relations. In detail, a state I consisting of terminated processes, all with the same continuation E, can evolve to p only if some processes of E, namely dec(E)-I', can do the same, leaving a set I' idle. As a consequence, all the processes in I; E cause the process p through the action a. Moreover, all the processes in I; E directly cause also those in I' which where enabled to perform actions independently of the processes in dec(E)-I'. Additionally, the rule expresses the fact that the processes in $\mathscr{R} \downarrow 2$ are still concurrent with those of dec(E)-I', and thus with a. Note also that predicate *end* is used, making sure that *all* the sequential processes of E_1 are terminated (thus they must be joined) and that *all of them* will cause the processes of E. Example 2.2.i) below reports an application of the second rule *Seq*). Rules *Conc*) allow concurrent sequential processes to proceed asynchronously, thus ensuring causal independency. Information about the context where concurrent processes operate is consistently reported also in relation $\mathscr{R}$ via tags "|id" or "id|". First rule *Dagger*) allows $E^\dagger$ to behave as E. Given a possible behaviour of $E^\dagger$, the remaining two rules recursively state that its behavior can be the same as that $E^\dagger \| E$ or $E \| E^\dagger$, too. This fact is assured by introducing the sequential processes of E as idles by means of relation $\mathscr{R}$. Note that n copies of E are put in parallel by a rewriting rule deduced with a proof in which the first rule *Dagger*) with dec(E)-$I' - [a, \mathscr{R}] \to p$ as premise has been used once, and then the second or third rule *Dagger*) have been used n-1 times (see Example 2.2.ii). As mentioned before, $E^\dagger$ can also behave like the empty process I, since end($\{E^\dagger\}$) is true. As regards rules *N*), let us recall that $\mathbf{N}(E_1, E_2, E_3, E_4)$ operationally behaves like $(E_1; E_3) \| (E_2; E_4)$ with the constraint that all the events of E_1 precede those of E_4. The first two rules are similar to *Conc*) rules and express the concurrency between E_1 and E_2.

Instead, the third and fourth rules are similar to the second *Seq)* rule in that they enable the processes of E_3 and E_4. When a process of E_3 starts, sequential process $I|\mathbf{N}$ is introduced to record termination of E_1. On the other hand, this very process is needed to enable E_4. The last rule permits the processes of E_4 to proceed independently of E_3 (they are concurrent), provided that E_1 and E_2 are completely evaluated; this requires the sequential processes in $\mathrm{dec}(E_3)|\mathrm{id}$ to be greater than those in $\langle I_1, E_3\rangle|\mathbf{N}$, but not than those in $\mathbf{N}|\langle I_2, E_4\rangle$. As a matter of fact, the $\mathbf{N}$-context fades in a $\|$-context, as soon as the processes of E_3 or E_4 are enabled. Example 2.2 may help to clarify how rewriting rules are deduced, and Example 2.3 shows how they will be used in actual computations.

Note that relation $\mathscr{R}$ is actually needed in rules *Seq)* and *N)*. As already mentioned, note also that only these rules deal with an implicit synchronization of many processes, which will form the set I of the rewriting rule $I - [a, \mathscr{R}] \to p$.

Example 2.2. We completely work out the derivations of two rewriting rules, the first of which is depicted in Fig. 3.

i) 1. $\{c\} - [c, \varnothing] \to I$ by *Act)*
2. $\{c|\mathrm{id}\} - [c, \varnothing] \to I|\mathrm{id}$ by 1 and *Conc)*
3. $\mathrm{end}(I)$ by Definition 2.4
4. $\mathrm{end}(\{I|\mathrm{id}, \mathrm{id}|I\})$ by 3 and Definition 2.4
5. $\{(I|\mathrm{id}; (c\|d), (\mathrm{id}|I); (c\|d)\} - [c, \{(I|\mathrm{id}); (c\|d), (\mathrm{id}|I); (c\|d)\} \leqslant \{\mathrm{id}|d\}] \to I|\mathrm{id}$
by 2 and 4 and second rule *Seq)*
(note that $\mathrm{dec}(c\|d) - \{\mathrm{id}|d\} = \{c|\mathrm{id}\}$)

ii) 1. $\{a\} - [a, \varnothing] \to I$ by *Act)*
2. $\{a^\dagger\} - [a, \varnothing] \to I$ by 1 and first rule *Dagger)*
3. $\{a^\dagger\} - [a, \{a^\dagger\} \leqslant \{a|\mathrm{id}\}] \to \mathrm{id}|I$ by 2 and second rule *Dagger)*
4. $\{a^\dagger\} - [a, \{a^\dagger\} \leqslant \{\mathrm{id}|a, (a|\mathrm{id})|\mathrm{id}\}] \to (\mathrm{id}|I)|\mathrm{id}$
by 3 and third rule *Dagger)*.

We now introduce our notion of computation as a finite sequence of states and rewriting rules. This notion is not the standard one, e.g., that of computation for transition systems, since we are dealing with rewriting systems which better reflect the asynchrony of our model.

Definition 2.6. (Computation) A sequence

$$\xi = \{J_0\ I_1 - [a_1, \mathscr{R}_1] \to p_1\ J_1\ I_2 - [a_2, \mathscr{R}_2] \to p_2$$
$$J_2 \cdots J_{n-1}\ I_n - [a_n, \mathscr{R}_n] \to p_n\ J_n\}$$

is a *computation* of E if

i) • $J_0 = \mathrm{dec}(E)$
   • J_i is a set of sequential processes, and
   • $I_i - [a_i, \mathscr{R}_i] \to p_i$ is a rewriting rule, $0 < i \leqslant n$;

ii) • $I_i \subseteq J_{i-1}$, and
   • $J_i = (J_{i-1} - I_i) \cup \mathscr{R}_i \downarrow 2 \cup \{p_i\}$, $0 < i \leqslant n$.

Computation ξ is *terminal* if $\mathrm{end}(J_n)$ holds.

Note that $\{\mathrm{dec}(E)\}$ represents an empty computation of E. When $E = 1$, $\{1\}$ is its only (terminal) computation; when $E = E_1^\dagger$, $\{E_1^\dagger\}$ is also a terminal computation of $E_1^\dagger$, since $\mathrm{end}(E_1^\dagger)$ holds. The following example

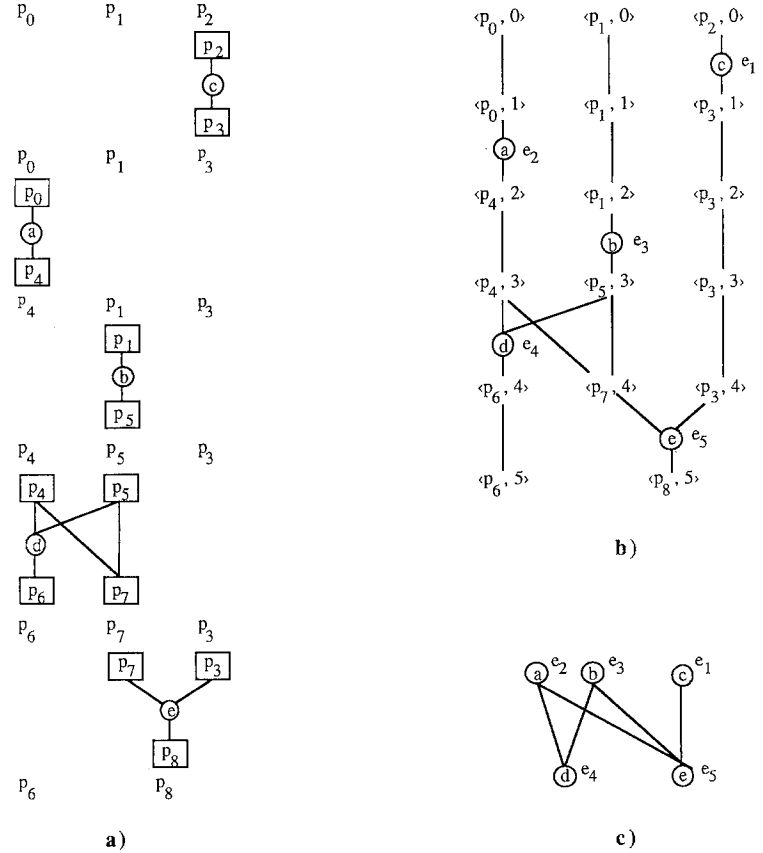

Fig. 4. A graphical representation of the computation ξ_1 of agent $\mathbf{N}(a \parallel b, c, d, e)$, given in Example 2.3.i (part a)); relation F^* (part b)) and the generated pomset (part c)), given in Example 2.4.i.

shows three instances of computation involving the "N" the "†", and the ";" operators; their pomsets will be built in Example 2.4, and shown on Figs. 4–6.

Example 2.3.

i) Given agent $E_1 = \mathbf{N}(a \| b, c, d, e)$, consider the sequential processes

$$p_0 = \langle a | \mathrm{id}, d \rangle | \mathbf{N} \qquad p_1 = \langle \mathrm{id} | b, d \rangle | \mathbf{N} \qquad p_2 = \mathbf{N} | \langle c, e \rangle$$
$$p_3 = \mathbf{N} | \langle 1, e \rangle \qquad p_4 = \langle 1 | \mathrm{id}, d \rangle | \mathbf{N} \qquad p_5 = \langle \mathrm{id} | 1, d \rangle | \mathbf{N}$$
$$p_6 = 1 | \mathrm{id} \qquad p_7 = 1 | \mathbf{N} \qquad p_8 = \mathrm{id} | 1.$$

From $\mathrm{dec}(E_1) = \{p_0, p_1, p_2\}$ the following terminal computation starts.

$$\xi_1 = \{\{p_0, p_1, p_2\}\ \{p_2\} - [c, \varnothing] \to p_3\ \{p_0, p_1, p_3\}\ \{p_0\} - [a, \varnothing] \to p_4$$
$$\{p_4, p_1, p_3\}\ \{p_1\} - [b, \varnothing] \to p_5\ \{p_4, p_5, p_3\}$$
$$\{p_4, p_5\} - [d, \{p_4, p_5\} \leqslant p_7] \to p_6\ \{p_6, p_7, p_3\}$$
$$\{p_7, p_3\} - [e, \varnothing] \to p_8\ \{p_6, p_8\}\}.$$

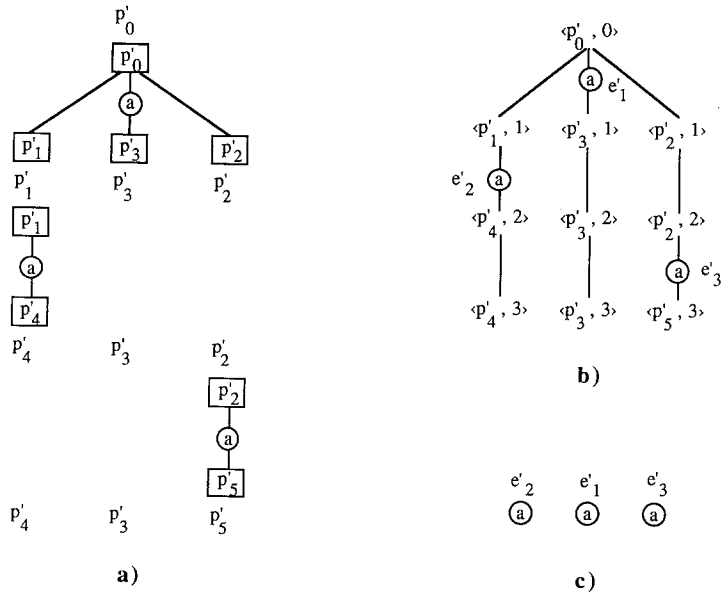

Fig. 5. A graphical representation of the computation ξ_2 of agent $a^{\dagger}$, given in Example 2.3.ii (part a)); relation F^* (part b)) and the generated pomset (part c)), given in Example 2.4.ii.

ii) Given agent $E_2 = a^\dagger$, consider the sequential processes.

$$p'_0 = a^\dagger \qquad p'_1 = (a|\mathrm{id})|\mathrm{id} \qquad p'_2 = \mathrm{id}|a$$

$$p'_3 = (\mathrm{id}|1)|\mathrm{id} \qquad p'_4 = (1|\mathrm{id})|\mathrm{id} \qquad p'_5 = \mathrm{id}|1$$

We have that from $\mathrm{dec}(E_2) = \{p'_0\}$ the following terminal computation starts.

$$\xi_2 = \{\{p'_0\}\ \{p'_0\} - [a, \{p'_0\} \leqslant \{p'_1, p'_2\}] \to p'_3\ \{p'_1, p'_3, p'_2\}$$
$$\{p'_1\} - [a, \varnothing] \to p'_4\ \{p'_4, p'_3, p'_2\}$$
$$\{p'_2\} - [a, \varnothing] \to p'_5\ \{p'_4, p'_3, p'_5\}\}.$$

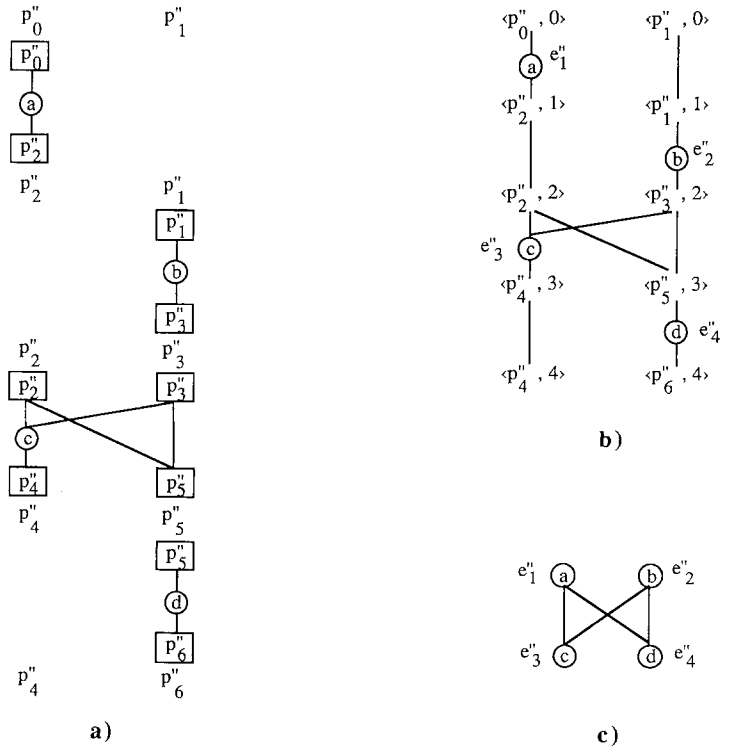

Fig. 6. A graphical representation of the computation ξ_3 of agent $(a \| b)$; $(c \| d)$, given in Example 2.3.iii (part a)); relation F^* (part b)) and the generated pomset (part c)), given in Example 2.4.iii.

iii) Given agent $E_3 = (a \| b); (c \| d)$, consider the sequential processes.

$$p_0'' = a|\mathrm{id}; (c \| d) \qquad p_1'' = \mathrm{id}|b; (c \| d) \qquad p_2'' = 1|\mathrm{id}; (c \| d)$$
$$p_3'' = \mathrm{id}|1; (c \| d) \qquad p_4'' = 1|\mathrm{id} \qquad p_5'' = \mathrm{id}|d$$
$$p_6'' = \mathrm{id}|1$$

We have that from $\mathrm{dec}(E_3) = \{p_0'', p_1''\}$ the following terminal computation starts.

$$\xi_2 = \{\{p_0'', p_1''\}\ \{p_0''\} - [a, \varnothing] \to p_2''\ \{p_2'', p_1''\}\ \{p_1'\} - [b, \varnothing] \to p_3''\ \{p_2'', p_3''\}$$
$$\{p_2'', p_3''\} - [c, \{p_2'', p_3''\} \leqslant \{p_5''\}] \to p_4''\ \{p_4'', p_5''\}$$
$$\{p_5''\} - [d, \varnothing] \to p_6''\ \{p_4'', p_6''\}\}.$$

From a computation, which is inherently sequential, we extract its pomset that records all the causal dependencies among its events. Essentially, this is done as follows: first an event e labelled by a is associated with every rewriting rule $I - [a, \mathscr{R}] \to p$ with the obvious causal dependencies between the processes in I, the new event and the process p, and those causal dependencies expressed by $\mathscr{R}$; then this relation is reflexively and transitively closed; finally all processes are removed to get the wanted pomset. We first recall the notion of pomset.[1]

Definition 2.7. (Pomset) A labelled partial ordering is a 4-tuple $\langle V, \Sigma, \leqslant, \mu \rangle$, where

- V is the vertex set of *events*
- Σ is the alphabet of *actions*
- $\leqslant$ is a partial ordering relation on V, called the *causal relation*
- $\mu: V \to \Sigma$ is the *labelling function*

Two events e_1 and e_2 are *concurrent* if neither $e_1 \leqslant e_2$ nor $e_2 \leqslant e_1$.
Two labelled orderings of events are *isomorphic* if there exists a label- and order-preserving bijection between their events.
A *pomset* (partially ordered multiset) $h = [V, \Sigma, \leqslant, \mu]$ is the isomorphism class of a labelled partial ordering.

Definition 2.8. (Generating pomsets) Given a terminal computation of agent E

$$\xi = \{J_0\ I_1 - [a_1, \mathscr{R}_1] \to p_1\ J_1\ I_2 - [a_2, \mathscr{R}_2] \to p_2$$
$$J_2 \cdots J_{n-1}\ I_n - [a_n, \mathscr{R}_n] \to p_n\ J_n\}$$

the pomset $[V, \Sigma, \leqslant, \mu]$ *generated via* ξ, denoted by $[E]_\xi$, is defined as follows.

i) Let $V = \{e_1, ..., e_n\}$ and $B = \{\langle p, i\rangle \mid p \in J_i\}$;

ii) Let F^* be the reflexive and transitive closure of relation F defined on $V \cup B$ by the following inference rules

- $p \in J_{i-1} - I_i$ **implies** $\langle p, i-1\rangle \; F \; \langle p, i\rangle$
- $p \in I_i$ **implies** $\langle p, i-1\rangle \; F \; e_i$
- $p \in (J_i - (J_{i-1} - I_i)) - \mathscr{R}_i \downarrow 2$ **implies** $e_i \; F \; \langle p, i\rangle$
- $\langle p_1, p_2\rangle \in \mathscr{R}_i$ **implies** $\langle p_1, i-1\rangle \; F \; \langle p_2, i\rangle$;

iii) Let $\Sigma = \{a_i\}$; $\leqslant$ be the restriction of F^* to S; and $\mu(e_i) = a_i$.

We now briefly comment on this construction. In order to obtain the pomset generated by a computation ξ two sets are constructed, the first consisting of events, the second of instances of processes, and then we determine the orderings over them. Index i in $\langle p, i\rangle$ is used to create a fresh instance of the process p which occurs in the distributed state J_i. The link between event e_i and the ith step of the computation $I_i - [a_1, \mathscr{R}_i] \rightarrow p_i$ is crucial for determining the causal ordering $\leqslant$. First, an auxiliary causal relation F^* between process instances and events is set by closing reflexively and transitively the causal ordering of the rewriting rules. More in detail, the first inference rule relates the two instances $\langle p, i-1\rangle$ and $\langle p, i\rangle$ of the same process p, which is idle in the ith step of the computation since it belongs to $J_{i-1} - I_i = J_i - (\mathscr{R}_i \downarrow 2 \cup \{p_i\})$. The second rule makes a process p smaller in F than the actual event e_i it performs; the third one makes this event smaller than the process p produced by this ith computation step. The last rule takes into account relation $\mathscr{R}$ for establishing the causal dependencies between a process $p \in I_i$ and those processes generated by p, but not by the event e_i. Note that making instances of the processes which occur in different states is crucial since it guarantees that relation F^* is acyclic, and makes it correctly mirror the flow of time. Eventually, the pomset generated by ξ is obtained by keeping only the events, by labelling them in the obvious way, and by restricting the causal ordering accordingly. Example 2.4 illustrates this construction.

Example 2.4. Given the agents and the computations of Example 2.3, we have that

i) $[E_1]_{\xi_1} = [V_1, \Sigma_1, \leqslant_1, \mu_1]$, where

- $V_1 = \{e_1, e_2, e_3, e_4, e_5\}$; • $\Sigma_1 = \{a, b, c, d, e\}$;
- $\{e_1 \leqslant_1 e_5, e_2 \leqslant_1 e_5, e_3 \leqslant_1 e_5, e_2 \leqslant_1 e_4, e_3 \leqslant_1 e_4, e_i \leqslant_1 e_i\}$;
- $\mu_1(e_1) = c$, $\mu_1(e_2) = a$, $\mu_1(e_3) = b$, $\mu_1(e_4) = d$, $\mu_1(e_5) = e$.

ii) $[E_2]_{\xi_2} = [V_2, \Sigma_2, \leqslant_2, \mu_2]$, where

- $V_2 = \{e'_1, e'_2, e'_3\}$;
- $\Sigma_2 = \{a\}$;
- $\{e'_i \leqslant_2 e'_i\}$;
- $\mu_2(e'_1) = \mu_2(e'_2) = \mu_2(e'_3) = a$.

iii) $[E_3]_{\xi_3} = [V_3, \Sigma_3, \leqslant_3, \mu_3]$, where

- $V_3 = \{e''_1, e''_2, e''_3, e''_4\}$;
- $\Sigma_3 = \{a, b, c, d\}$;
- $\{e''_1 \leqslant_3 e''_3, e''_1 \leqslant_3 e''_4, e''_2 \leqslant_3 e''_3, e''_2 \leqslant_3 e''_4, e''_i \leqslant_3 e''_i\}$;
- $\mu_3(e''_1) = a, \mu_3(e''_2) = b, \mu_3(e''_3) = c, \mu_3(e''_4) = d$.

Figures 4–6 show computations ξ_1, ξ_2 and ξ_3, an intermediate step of the construction of Definition 2.8 after having determined F^*, and the generated pomsets.

In the pomset represented in Fig. 4), event e_4 (labelled by d) has been generated in correspondence to a rewriting rule temporally applied before the one corresponding to event e_5 (labelled by e), but it is easy to see that the two concurrent events could also be generated in the inverse temporal ordering. It suffices to substitute the last two steps of the computation ξ_1 with the following

$$\{p_4, p_5, p_3\}\ \{p_4, p_5, p_3\} - [e, \{p_4, p_5\} \leqslant \{p'_6\}] \to p_8$$
$$\{p'_6, p_8\}\ \{p'_6\} - [d, \varnothing] \to p_6\ \{p_6, p_8\}$$

where $p'_6 = d|\mathrm{id}$, and the first rewriting rule is obtained via the last ***N***) inference rule.

This fact holds in general, since the rewriting system of Definition 2.5 is *completely concurrent*, in the terminology used in Ref. 16. Roughly speaking, complete concurrency says that, given two concurrent events in the pomset $[E]_\xi$ (generated by computation ξ), there always exists another computation ξ' where the two events are generated in inverse ordering, and such that $[E]_\xi = [E]_{\xi'}$. In other words, we can say that all and only the linearizations of a partial ordering are induced by computations. An immediate consequence is that the operation of *linearization* introduced by Pratt is implicitly present in our operational semantics: it suffices to let event e_i be smaller than e_j if $i < j$. The proof that the rewriting system of Definition 2.5 is completely concurrent is long and outside the scope of this paper; we only note that two symmetrical rules *Dagger*) (the second and third) have been introduced to this purpose, in place of a single one, encompassing both.

We finally define the operational semantics of an agent E as the set of pomsets generated via the terminal computations of E. Even if there is no explicit nondeterministic operator, the simple language defined is nondeterministic, because of operator "†".

Definition 2.9. (Operational semantics) Given a term E, its *operational semantics* is defined as

$$[E]_o = \{[E]_\xi \mid \xi \text{ is a terminal computation of } E\}.$$

The operational semantics for the basic language we consider here has been defined through a set of inference rules which are driven by the syntactic structure of terms, and thus it is compositional, a crucial property that any semantic definition must enjoy. We also remark that compositionality provides us with the means for easily proving in the next section that the denotational semantics of the language is fully abstract with respect to the operational one.

3. EQUIVALENCE OF OPERATIONAL AND DENOTATIONAL SEMANTICS

The operational semantics defined in the previous section formally describes the behavior of concurrent systems, mimicking the intuition behind their dynamics. Denotational semantics is more abstract and complementary to operational semantics in that it takes no account of any implementation issues. Moreover, it makes reasoning about systems easier, since it is compositional and properties of the semantic domain can be exploited. Of course, the two semantics must agree, and their equivalence must be formally established, so that either semantics can be properly used. We will show that there is quite a strong equivalence between the semantics of the simple language considered here. First, we will recall Pratt's definitions of the needed semantic operators on pomset, through which we define the denotational semantics of the language.

Definition 3.1. (Pratt's operators on pomsets.) Let $h_i = [V_i, \Sigma_i, \leqslant_i, \mu_i]$, $V_i \cap V_j = \varnothing$, $i \neq j\, 1 \leqslant i, j \leqslant 4$, be four pomsets; we consider the following operators on them, which are understood to operate elementwise, when applied to sets.

;) $h_1; h_2 = [V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, \leqslant_1 \cup \leqslant_2 \cup V_1 \times V_2, \mu_1 \cup \mu_2]$;

||) $h_1 \| h_2 = [V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, \leqslant_1 \cup \leqslant_2, \mu_1 \cup \mu_2]$;

†) $h_1^\dagger = \bigcup_{i \in \mathrm{Nat}} (h_1)^i$, where $h_1^0 = [\varnothing, \varnothing, \varnothing, \varnothing]$ and $(h_1)^{i+1} = h_1 \| (h_1)^i$;

N) $\mathbf{N}(h_1, h_2, h_3, h_4) = [V_1 \cup V_2 \cup V_3 \cup V_4, \Sigma_1 \cup \Sigma_2 \cup \Sigma_3 \cup \Sigma_4,$
$\leqslant_1 \cup \leqslant_2 \cup \leqslant_3 \cup \leqslant_4 \cup V_1 \times V_3 \cup V_2 \times V_4 \cup V_1 \times V_4,$
$\mu_1 \cup \mu_2 \cup \mu_3 \cup \mu_4]$.

The intuition underlying these semantic operators is explained as follows. Two pomsets are sequentially concatenated through ";" resulting in a pomset where all the events of the first pomset precede all the events of the second one. Concurrently composing two pomsets via "||" means letting them lie side by side. Operator "†" generates a set of pomsets obtained by concurrently composing finitely many copies of the given pomset. Finally, four pomsets are composed into a N-shaped pomset via operator "N". Note that operator "†" is well-defined since the power of pomsets is continuous, as well as all the other operators, on sets of pomsets ordered by inclusion. Note also that all the operators are strict.

The denotational semantics of the simple language introduced in Section 2 can now be easily defined by inducing on the structure of terms. It reflects the intuitive description of agents given after Definition 2.1. As was the case for operational semantics, also the denotational semantics of a term turns out to be a set of pomsets, due to the operator "†".

Definition 3.2. (Denotational semantics) Given a term E, its denotational semantics, denoted by $[E]_d$, is defined by structural induction as follows.

$$[1]_d = \{[\varnothing, \varnothing, \varnothing, \varnothing]\}; \qquad [a]_d = \{[\{e\}, \{a\}, \{e \leqslant e\}, \mu(e) = a]\};$$

$$[E_1; E_2]_d = [E_1]_d; [E_2]_d; \qquad [E_1 \| E_2]_d = [E_1]_d \| [E_2]_d;$$

$$[E^\dagger]_d = [E]_d^\dagger; \qquad [\mathbf{N}(E_1, E_2, E_3, E_4)]_d = N([E_1]_d, [E_2]_d, [E_3]_d, [E_4]_d).$$

The next theorem will show that the denotational and operational semantics coincide.

Theorem 3.1. (Operational semantics = denotational semantics) For every agent E, $[E]_d = [E]_o$.

Proof. It suffices to prove by structural induction that, given an agent E, a pomset h belongs to $[E]_o$ if and only if it belongs to $[E]_d$. The base cases are when the agent is either 1 or a, and the proof is trivial. We now proceed by case analysis, inductively assuming that $[E_i]_d = [E_i]_o$.

$\boldsymbol{E = E_1; E_2}$.

if-part) $h \in [E_1; E_2]_d$ implies $h \in [E_1; E_2]_o$.

Before proving the claim, we introduce an operation ";E" which transforms a given computation

$$\xi = \{\mathrm{dec}(E')\ r_1\ J_1 \cdots J_{n-1}\ r_n\ J_n\}$$

in the following computation

$$\xi; E_2 = \{\mathrm{dec}(E'); E\ r_1; E\ J_1; E \cdots J_{n-1}; E\ r_n; E\ J_n; E\},$$

where $r_i; E$ is obtained with first rule *Seq*) with premise r_i and continuation E. More precisely, given a rewriting rule $r_i = I - [a, \mathscr{R}] \to p$, then $r_i; E = I; E - [a, \mathscr{R}; E] \to p; E$.
By inductive hypothesis we know that there exist two terminal computations

$$\xi_1 = \{\mathrm{dec}(E_1)\ r_1\ J_1 \cdots J_{n-1}\ r_n\ J_n\}$$

$$\xi_2 = \{\mathrm{dec}(E_2)\ \mathrm{dec}(E_2)\text{-}I - [a, \mathscr{R}] \to p\ J'_1 \cdots J'_{m-1}\ r'_m\ J'_m\}$$

which generate pomsets h_i belonging to $[E_i]_{\mathscr{A}} = [E_i]_{\mathscr{O}}$, $i = 1, 2$.
Since ξ_1 is terminal, $\mathrm{end}(J_n)$ holds. From it and $\mathrm{dec}(E_2)\text{-}I - [a, \mathscr{R}] \to p$ (i.e. from the first rewriting rule of ξ_2) we can deduce, by using the second rule *Seq*), the rewriting rule
$r = J_n; E_2 - [a, J_n; E_2 \leqslant \mathscr{R} \downarrow 2 \cup I] \to p$. It is easy to see that computation

$$\xi = \{\mathrm{dec}(E_1); E_2\ r_1; E_2\ J_1; E_2 \cdots J_{n-1}; E_2\ r_n; E_2\ J_n; E_2$$
$$r\ J'_1 \cdots J'_{m-1}\ r'_m\ J'_m\}$$

originates exactly $h = h_1; h_2$, since all the sequential processes of $J_n; E_2$ dominate in rule r (either via a or relation $J_n; E_2 \leqslant \mathscr{R} \downarrow 2 \cup I$) all the sequential processes of J'_1.
only if-part) $h \in [E_1; E_2]_{\mathscr{O}}$ implies $h \in [E_1; E_2]_{\mathscr{A}}$.
We have a terminal computation ξ for $E_1; E_2$ generating h,

$$\xi = \{\mathrm{dec}(E_1); E_2\ r_1; E_2\ J_1; E_2 \cdots J_{n-1}; E_2\ r_n; E_2\ J_n; E_2$$
$$r\ J'_1 \cdots J'_{m-1}\ r'_m\ J'_m\}.$$

We certainly have that $\mathrm{end}(J_n)$, because all the sequential processes produced by E_1 must terminate in order to enable those of E_2. Thus, we can extract from ξ, the following terminal computation for E_1

$$\xi_1 = \{\mathrm{dec}(E_1)\ r_1\ J_1 \cdots J_{n-1}\ r_n\ J_n\}$$

which generates a pomset h_1 which, by inductive hypothesis is such that $h_1 \in [E_1]_{\mathscr{A}}$. Also, we have that rule r in ξ has the form $J_n; E_2 - [a, J_n; E_2 \leqslant \mathscr{R} \downarrow 2 \cup I] \to p$ since it must have been obtained through the second rule *Seq*) with premises $\mathrm{end}(J_n)$ and $\mathrm{dec}(E_2)\text{-}I - [a, \mathscr{R}] \to p$. Now, it is easy to construct the following computation

$$\xi_2 = \{\mathrm{dec}(E_2)\ \mathrm{dec}(E_2)\text{-}I - [a, \mathscr{R}] \to p\ J'_1 \cdots J'_{m-1}\ r'_m\ J'_m\}$$

which is terminal for E_2, since $\mathrm{end}(J'_m)$ holds (recall that ξ is terminal) and generates, by inductive hypothesis, the pomset $h_2 \in [E_2]_{\mathscr{A}}$. The claim

follows by noting that all the events of h_1 precede in h those of h_2, thus $h = h_1; h_2$.

$E = E_1 \| E_2$.
if-part) $h \in [E_1 \| E_2]_{\mathcal{A}}$ implies $h \in [E_1 \| E_2]_{\mathcal{O}}$.
The proof is analogous to the previous one, provided that computations ξ_1 and ξ_2 are modified by adding tags "|id" and "id|", respectively. Note also that the sequential processes of the two original computations are made distinct in the new computation obtained in this way, and thus the actions they perform will never be related by the generated partial ordering.
only if-part) $h \in [E_1 \| E_2]_{\mathcal{O}}$ implies $h \in [E_1 \| E_2]_{\mathcal{A}}$.
Again the proof is analogous to the one given for ";". One has to notice that from the given computation of $E_1 \| E_2$, two computations of E_1 and E_2 can be derived. Each distributed state of the former can easily be split in two subsets consisting of all sequential processes with "|id" or "id|" as outermost tags. These subsets, provided that the outermost tags are removed, correspond to each distributed state of the required computations of E_1 and E_2. The required rewriting rules of these computations can easily be obtained in a similar way.

$E = E^{\dagger}$.
if-part) $h \in [E^{\dagger}]_{\mathcal{A}}$ implies $h \in [E^{\dagger}]_{\mathcal{O}}$.
A further induction is needed on the number of copies of E which are put in parallel in h. Recall that we are dealing with finitely many copies, by definition, thus Peano's induction suffices. The number n of copies of E is related to which of the rules *Dagger*) has been used in deducing the appropriate rewriting rule, and, in particular, to the number of times that the second and third rule *Dagger*) have been used. When $n = 0$, i.e., when $h = [\varnothing, \varnothing, \varnothing, \varnothing]$, the claim holds since $\{E^{\dagger}\}$ is the required terminal computation originating the empty pomset. The outermost inductive hypothesis suffice when $n = 1$, i.e. when $h \in [E]_{\mathcal{A}}$. Indeed, the first rewriting rule of every nonempty computation of E has been generated by using once the first rule *Dagger*). Also the inductive step is routine: if we have n copies of one of the pomsets of E in h, the second (third) rule *Dagger*) has been used $n-1$ times. In fact, each of these application generates a new copy of dec(E), enriched with tag "|id" ("id|"), and we are reduced to the case "||".
only if-part) Follows by noting that rule *Dagger*) only generates finitely many (disjoint) copies of a pomset generated by an agent E.

$E = \mathbf{N}(E_1, E_2, E_3, E_4)$.
if-part) $h \in [\mathbf{N}(E_1, E_2, E_3, E_4)]_{\mathcal{A}}$ implies $h \in [\mathbf{N}(E_1, E_2, E_3, E_4)]_{\mathcal{O}}$.
A computation for E is obtained by first modifying the computations ξ_1, ξ_2, ξ_4, ξ_3 of E_1, E_2, E_4, E_3 to record the appropriate context, and then by

concatenating them in the given order. Concatenating computations ξ_1 and ξ_2, and ξ_4 and ξ_3 is done as in the case of "||". In order to concatenate ξ_2 and ξ_4, the fourth inference rule N) has to be used. Actually, its premises ensure that ξ_1 and ξ_2 are terminated and relation $\mathscr{R}_1$ is such that all the events of computations ξ_1 and ξ_2 are set smaller than those of ξ_4. Note also that the same rule lets all the events of computations ξ_1 be smaller than those of ξ_3 because of relation $\mathscr{R}_2$.

only if-part) Just reverse the argument and take care of tags similarly to the case of "||". ∎

A straightforward consequence of the compostionality of the operational and of the denotational semantics is that the latter is fully abstract with respect to the former.[26] In other words, the equivalence, rather the congruence relation induced by the operational semantics coincides with that induced by the denotational semantics. Furthermore this congruence proves to be the minimal one. We need the notion of context $C[\,]$ which is, as usual, a term with one or more holes to be filled by an agent. A possible context is a term such as $E \| \bullet$, where $\bullet$ is a hole. When an agent, say E', is substituted for $\bullet$, we obtain the agent $E \| E'$. Another example is context $(E \| \bullet)$; $\bullet$ which may become agent $(E \| E')$; E'. Now we can state our fully abstractness corollary which follows immediately from the equivalence theorem and from the so-called *fully abstractness in se*[26] of the denotational semantics. This property amounts to saying that, for every context $C[\,]$, $[E_1]_{\mathscr{A}} = [E_2]_{\mathscr{A}}$ if and only if $[C[E_1]]_{\mathscr{A}} = [C[E_2]]_{\mathscr{A}}$, which is in our case obvious.

Corollary 3.1. (Denotational semantics is fully abstract w.r.t. operational semantics) For every context $C[\,]$, $[E_1]_{\mathscr{A}} = [E_2]_{\mathscr{A}}$ if and only if $[C[E_1]]_o = [C[E_2]]_o$.

CONCLUSION

Our claim is that the semantics of concurrent languages is better defined within the true concurrency approach, where causal dependencies and independencies among the events performed by a concurrent system are explicitly represented through partial orderings. We also think that both an operational and a denotational semantics should be given and proved equivalent, the former for making intuition formal, the latter for making reasoning easier. Some languages have already been provided with a partial ordering operational semantics, but only classical operators for concurrency and sequentialization (plus nondeterminism) have been dealt with in the true concurrency approach.[2,12,13,15,16,23–25,27] Here for the first time, we have shown that a partial ordering based semantics can be

operationally defined also for concurrent languages with operators like N which are not expressible through the standard series/parallel ones, only. To the best of our knowledge, this operator has been given a denotational semantics only in Ref. 1, from which we have also taken other basic operators.

The goal has been achieved by using labelled rewriting systems that describe how parts of a distributed state evolve, rather than more classical transition systems that relate global states. Thus, the abstract machine we use for defining the operational semantics needs a great deal of detail to explicitly express the dependencies and independencies of spatially distributed events which are performed by concurrent systems. Thus, implementation issues had to be taken into account in defining our (abstract) distributed interpreter, which adds further complexity to the more standard interleaving oriented transition systems. We remark that other models within the partial ordering approach are not completely satisfactory as a basis for giving a fully satisfactory true concurrent operational semantics. For instance, as they are, Petri Nets are not compositional, and this fundamental property plays a crucial role in the present work.

The paradigm we have followed in defining the operational semantics of our simple language is as follows. Given a term, representing a global state of a concurrent system, we have defined its distributed state as the set of its sequential processes, i.e., its simplest sub-terms that may perform actions independently. Then we have defined, in the SOS style, a set of rewriting rules which express the dynamics of a system by relating only those sequential processes that actually evolve. Non-conflicting rewriting rules can, in principle, be applied in parallel to sequential processes. Instead, we have represented a distributed computation as a sequence of (states and of) rules; nevertheless, the use of a sequential interpreter in place of a parallel one is only a simplification and does not affect the essence of the model. Actually, our rewriting system has the property of complete concurrency,[16] stating that two concurrent events can be generated in either temporal order. The operational meaning of a term is eventually generated by abstracting from the interleavings forced by the sequential interpreter and consists of a set of pomsets containing the performed events and the complete causal relations set up in the computation. Finally, a denotational semantics has been defined in terms of Pratt's combinatorial operators on pomsets.

We have proved that the denotational semantics is strongly equivalent to our operational semantics, in that they coincide and, additionally, in that the latter is fully abstract with respect to the former. To the best of our knowledge, such a result has been proved here for the first time in the true

concurrency framework, even if it holds for an admittedly simple language, and we are confident that adding other operators will not affect it. We remark that all the proofs are carried out by structural induction and, though long and tedious, are straightforward because also our operational semantics is compositional.

All the languages proposed so far have been developed within the interleaving approach, and thus their operators are strongly based on the series/parallel ones. These languages pay no attention to non-classical operators, like the **N** operator considered here. Note however that **N**-shaped pomsets could be obtained in this framework, with an obscure construction which resorts to auxiliary actions, synchronization and hiding, e.g., the TCSP[5] term $((a;(c \|_{\varnothing} e)) \|_{\{e\}} ((e \|_{\varnothing} b);d))\backslash e$, where e is an auxiliary action on which first synchronize and then restrict upon, originates the pomset of Figure 1a. Certainly, the interleaving approach does not call for such operators, or for any generalizations of them, or even for operators which make sense only when more complex semantic domains, with a richer structure than sets of pomsets, are considered, e.g., event structures[18,21] in which concurrency, nondeterminism, causality and mutual exclusion originate intricate interplays. Thus, the problem arises of determining a minimal set of expressive operators, if any, through which it will be possible to define all the partial orderings meaningful in a specific semantic domain. Nonetheless, whichever operator is introduced, provided that it has an operational intuition, we are confident that the approach followed here will suffice to describe its behavior: decomposition functions and rewriting systems or related formalisms such as Petri Nets[16,23,24,27] seem to be powerful enough for defining operationally the semantics of concurrent systems in terms of partial orderings, just as transition systems are the basis for giving operational semantics to sequential languages, and for defining interleaving models for concurrency.

ACKNOWLEDGMENTS

We wish to thank Roberto Gorrieri and three anonymous referees for their detailed comments and helpful suggestions. LIST supported the second author in a friendly and pleasant environment during the revision of the paper.

REFERENCES

1. V. Pratt, Modelling Concurrency with Partial Orders, *International Journal of Parallel Programming*, **15**:33–71 (1986).
2. P. Degano, R. De Nicola, and U. Montanari, A Partial Ordering Semantics for CCS. Dipartimento di Informatica Research Rep TR-3/88.

3. D. Austry and G. Boudol, Algèbre de Processus et Synchronization, *Theoret. Comput. Sci.*, **30**(1):91–131 (1984).
4. J. A. Bergstra and J.-W. Klop, Process Algebra for Synchronous Communication, *Info. and Co.*, **61**:109–137 (1984).
5. S. D. Brookes, C. A. R. Hoare, and A. D. Roscoe, A Theory of Communicating Sequential Processes, *Journal of ACM*, **31** (3):560–599 (1984).
6. M. Hennessy, *An Algebraic Theory of Processes*, MIT Press, (to appear).
7. L. Lamport, What Good is Temporal Logic? *Proc. IFIP* '83, North-Holland, Amsterdam, pp. 657–668 (1983).
8. G. Milne, CIRCAL and the Representation of Communication, Concurrency and Time, *ACM TOPLAS*, 7(2):270–298 (1985).
9. R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol. 92, Springer–Verlag, Heidelberg (1980).
10. M. Nivat, Behaviours of Processes and Synchronized Systems of Processes, in: *Theoretical Foundations of Programming Methodology* M. Broy and G. Schmidt (eds.), Reidel, Dodrecht, pp. 473–550 (1982).
11. G. Plotkin, A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, Department of Computer Science, Aarhus, (1981).
12. G. Boudol and I. Castellani, On the Semantics of Concurrency, Partial Orders and Transition Systems, in *Proc. Tapsoft-CAAP '87*, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, **249**:123–137 (1986).
13. M. Broy and T. Steicher, Views of Distributed Systems, Proc. Advanced School on Mathematical Models for the Semantics of Parallelism. Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, **280**:114–143 (1987).
14. Ph. Darondeau and L. Kott, On the Observational Semantics of Fair Parallelism, in Proc. ICALP, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 147–151 (1983).
15. P. Degano and U. Montanari, A Model of Distributed Systems Based on Graph Rewriting, *Journal of ACM*, **34**:411–449 (1987).
16. P. Degano and U. Montanari, Concurrent Histories: A Basis for Observing Distributed Systems, *Journal of Computer and System Sciences*, **34**:442–461 (1987).
17. L. Lamport, Time, Clocks and the Ordering of Events in a Distributed System, *Communication of ACM*, **12**:558–564 (1978).
18. M. Nielsen, G. Plotkin, G. Winskel, Petri Nets, Event Structures and Domains, Part 1, *Theoret. Comput. Sci.*, **13**:85–108 (1981).
19. C. A. Petri, Concurrency, in *Net Theory and Applications*, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, **84**:1–19 (1980).
20. J. Winkowski, Behaviours of Concurrent Systems, *Theoretical Computer Science* **12**:39–60 (1980).
21. G. Winskel, Petri Nets, Algebras, Morphisms and Compositionality, *Info. and Co.*, **72**:197–238 (1987).
22. U. Goltz and W. Reisig, The Non-sequential Behaviour of Petri Nets, *Info. and Co.* **57**:125–147 (1983).
23. P. Degano, R. De Nicola, and U. Montanari, CCS is an (Augmented) Contact-Free Condition/Event System, Proc. Advanced School on Mathematical Models for the Semantics of Parallelism. Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, **280**:144–165 (1987).
24. P. Degano, R. Nicola, and U. Montanari, A Distributed Operational Semantics for CCS based on Condition/Event Systems. Nota Interna B4-21 (IEI, 1987). (to appear in *Acta Informatica*).

25. P. Degano, R. Gorrieri, and S. Marchetti, An Exercise in Concurrency: A CSP Process as a C/E System, in *Advances in Petri Nets 1988*, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg (1988) (to appear).
26. R. Milner, Fully Abstract Models for Typed Lambda-Calculi, *Theoret. Comput. Sci.*, **4**:1–23 (1977).
27. E.-R. Olderog, Operational Petri Net Semantics for CCSP. In *Advances in Petri Nets 1987*, G. Rozenberg, (ed.) Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, **266**:196–223 (1987).
28. A. Mazurkiewicz, Concurrent Program Schemas and Their Interpretation, Proc. Aarhus Workshop on Verification of Parallel Programs, (1977).