Theoretical Computer Science 59 (1988) 25-84
North-Holland

CONCURRENCY AND ATOMICITY

G. BOUDOL and I. CASTELLANI
INRIA Sophia-Antipolis, 06560 Valbonne Cedex, France

Abstract. The overall intention of this work is to investigate the ability to regard a finite computation as a single event, in dealing with the semantics of concurrency. We propose a calculus of concurrent processes that embodies this ability in two respects: the first one is that of execution, the second that of operation. As usual, we formalize the execution of a process as a labelled transition relation. But our point is that at each step the performed action is a compound one, namely a labelled poset, not just an atom. The action reflects the causal and concurrent structure of the process, and we claim that the bisimulation relative to such transition systems brings out a clear distinction between concurrency and sequential nondeterminism. Next we introduce a second transition relation, formalizing the operation of a process on data. As in the usual semantics of sequential programs, a process operates on data by means of its terminated sequences of computations. Then we obtain atomic actions by abstracting the whole operation of a process as a single event. We show that this abstraction mechanism, together with the idea of compound actions, allows us to deal with a variety of synchronization and communication disciplines.

1. Introduction

The purpose of this paper is twofold: to set up a semantics for "true concurrency" and to propose a semantics for atomic actions. In developing our proposals we shall use ideas borrowed from various approaches to the semantics of concurrency. Milner's work on *calculi of processes* [44, 46, 47] provides our main source of inspiration. Let us recall the main features of such calculi (cf. [5]): first there is a *syntax* which describes *abstract programs* as terms of an algebra; second there are *behavioural rules* according to which each term may perform some *actions* and become another term in doing so. This brings in a notion of *labelled transitions:*

$$process \xrightarrow[\text{execution}]{action} process'.$$

Then a *semantic equality* is defined by means of the well-known notion of bisimulation [51, 46, 8]. This semantic equality usually complies with algebraic laws, which may form the basis for an axiomatization.

A common feature of process calculi is that they are based on operational semantics of *programming constructs.* This is precisely what we are interested in. One of the reasons is the following: although there is some progress towards a Church's Thesis for concurrency (cf. [70, 63, 5, 48]), we still lack a mathematical notion of process which could play a rôle similar to that of (recursive) function for sequential languages. A Church's Thesis would emerge from a collection of results establishing

0304-3975/88/$3.50 © 1988, Elsevier Science Publishers B.V. (North-Holland)

the equivalence of various models. As regards concurrency, the available results of this kind are not based upon a universally accepted notion of process. Then a possible preliminary step is to bring out some primitive constructs for "computing processes", and to compare them. This was an early purpose of CCS (cf. [43]). Later on, Milner took the standpoint [46, 47], which we adopt, that any abstract notion of process must be based firmly upon operational semantics.

Informally, we view a concurrent system as made out of autonomous agents which may communicate during their execution. This is just to point out the difference from sequential systems, which communicate at the end of their execution. In order to formalize such a notion of interactive and possibly nonterminating process, one must take into account concurrency, communication and execution. We shall return to the first two concepts later. As regards execution, transition systems provide us with a suitable starting point. As Plotkin shows in [53], a fairly general model of the operational semantics of programming languages is that of transitions inferred by means of structural rules. This also copes with languages for concurrent and communicating processes, such as CCS or CSP (cf. [54]).

Bisimulations on transition systems provide a powerful concept (see [63, 5, 8]), but many authors argue ([12, 16, 59], to mention but a few) that this yields an inadequate description of concurrency. Specifically what is questioned is Milner's expansion theorem [44, 30], expressing a simulation of concurrency by sequential nondeterminism. Roughly speaking, $(a|b) = ab + ba$ (we use here a CCS-like notation), thus the parallel composition operator can be eliminated (from finite terms), whence it is not primitive. As a contribution to the theory of "true concurrency", our paper aims to solve

$$\textit{concurrency} \neq \textit{sequentiality} + \textit{nondeterminism}. \tag{1}$$

More precisely our thesis is that this can be solved while still dealing with bisimulations on transition systems: we show that the criticism should not be moved against the expansion theorem, but rather against the lack of structure in the actions of transition systems.

Evolving from Petri's ideas [52], there is another way to approach the semantics of concurrency. Following this way one thinks of sequentiality as arising from *causal* structure which prescribes an ordering on events. Dually, two events are concurrent if they are not causally related. Thus an action is a *partially ordered set of events,* labelled by atoms, rather than a mere sequence. This is by now a widely held point of view, which we shall survey in a first section of the paper. Let us just say that our main source of inspiration from this area is Grabowski and Gischer's theory of *pomsets* [27, 24], which are partially ordered multisets of atoms. A step towards the idea of actions as pomsets was already taken in the calculus MEIJE [2, 5], where an action is a multiset, that is, a parallel product of atoms. We can claim that in MEIJE concurrency is primitive since we have

$$(a|b) = ab + (a|b) + ba \neq ab + ba.$$

However MEIJE does not provide an adequate model for causality since we have something like

$$(ab|c) = a(b|c) + (a|c)b.$$

These examples show that there is still a notion of global time in this calculus, which forces some undesirable causal dependencies. What we seek is an execution model where causality, as well as concurrency, is properly taken into account.

Let us now introduce our first contribution: first of all, in order to solve (1) we must start with a formalism in which one can talk about sequentiality, nondeterminism and concurrency as distinct notions. This is why we adopt as our *system model* Winskel's (labelled) event structures, built upon the exclusive relations of causal ordering, conflict and concurrency. Each of these relations gives rise to a way of constructing event structures: one simply juxtaposes two such structures and then sets the corresponding relation between their events. These operations are *sequential composition* (;), *sum* (+), and *parallel composition* (‖); they provide us with a syntax for finite event structures. Obviously, the syntactic distinction between the three operations is too crude: we would like to identify some terms on behavioural grounds.

We now describe the first main idea. An event structure determines a set of computations, what Winskel calls configurations: a computation is a conflict-free "prefix" of the event structure. We shall restrict our attention to finite computations. Then, defining "what remains of the structure" after a computation we get a notion of labelled transition: the action (= the computation) is a finite pomset and the state reached (= what remains...) is another event structure. This provides us with an *execution model*. The point is that we generalize what is usually over the (execution) arrow: computations are finite and determinate processes, not just single atoms. Syntactically they are denoted by terms involving sequential and parallel composition, but not sum.

We then give a structural operational semantics, following Plotkin's style [53, 54], for our "abstract programs", and show an exact correspondence between the semantical and syntactical notions of transition. Since computations may contain causality and concurrency, we shall have some rules introducing the corresponding operators over the arrow, namely:

$$p \xrightarrow{u} p'\dagger \;\&\; q \xrightarrow{v} q' \;\Rightarrow\; p;q \xrightarrow{u;v} q'$$

where $p'\dagger$ means that p' is a terminated program, and

$$p \xrightarrow{u} p' \;\&\; q \xrightarrow{v} q' \;\Rightarrow\; (p\|q) \xrightarrow{(u\|v)} (p'\|q').$$

Let us see just one example: we have

$$(a;b;p\|c;q) \xrightarrow{(a;b\|c)} (p\|q).$$

This shows that our calculus is *asynchronous*: there is no assumption of global time since two concurrent processes may independently perform computations of arbitrary lengths.

Next we define our semantic equality, in the same way as Milner defines his strong congruence, and give an axiomatization for it. We claim that this notion of equality solves (1); so this ends our proposed execution model for "true concurrency".

There is a remaining point in our informal view of a process—as a system made out of communicating concurrent agents—that of communication. We may roughly distinguish two kinds of communication: an *indirect* one, usually asynchronous, where the agents communicate by means of *shared objects* (variables, buffers, and so on); a *direct* one, usually synchronous, where the agents exchange messages. We can regard a direct communication act as a computation of the form $(u\|v)$ since it requires the participation of at least two concurrent partners. If the communication is synchronous, one must ensure that in some sense $(u\|v)$ cannot be dissociated. One way to achieve this is to *specify* it as an *axiom*, an interaction law as in Winskel's synchronization algebras [69]. For instance, for the calculi CCS, MEIJE/SCCS and TCSP, we would have something like

$$\begin{aligned} \text{CCS:} &\quad (a\|\bar{a}) = \tau, \\ \text{MEIJE/SCCS:} &\quad (a\|a^{-1}) = 1, \\ \text{TCSP:} &\quad (a\|a) = a. \end{aligned}$$

There is another way to ensure synchronization, which is to *implement* it in some sense. This is the way we shall follow. More specifically, we shall "implement" direct synchronous communication by means of indirect asynchronous one, using a formal notion of atomic action. This is the second main idea of the paper: we want to set up an operational formalism where it is possible to *abstract* the whole behaviour of a process as a single *atomic action*. In such a formalism, we shall be able to say that some "high-level" primitive is implemented at a more concrete level.

In order to introduce the abstraction mechanism, let us return for a while to communication: for indirect communication, the primitive act is the *application* of an action to an object. For instance, this can consist in reading or updating a variable, or in putting a value into a buffer, or getting a value from that buffer, and so on. We shall denote by $s \mapsto^{u} s'$ the fact that an action u, when applied to the object s, *operates* on it, changing it into s' (the last part of the paper deals with the notions of object and operation). Applying an action to an object may consist in applying an arbitrarily complex process, and it is the whole operation of such a process that will be abstracted as an atomic action. Therefore we have to define the operation of processes, not only of actions, on objects. This is formalized as a new transition relation:

$$\textit{object} \xmapsto[\text{operation}]{\textit{process}} \textit{object}'.$$

As a matter of fact, the inference rules for operation are fairly simple: they just say that a process operates by means of its *terminated sequences* of computations. This

operation may be nondeterministic, due to an arbitrary interleaving of the concurrent components of the process.

Atomic actions are then introduced within a definition similar to that of abstract data type, namely $o = (s \text{ with } a_1 = p_1, \ldots, a_k = p_k)$, where s is an object, the a_i's are atom identifiers and the p_i's their respective codes, which are processes. The term o is an (abstract) object, and the corresponding operation rule for it—that is, the *abstraction rule*—is, roughly speaking, as follows:

$$s \xrightarrow{u[p_i/a_i]} s' \vdash (s \text{ with } a_1 = p_1, \ldots, a_k = p_k) \overset{u}{\mapsto} (s' \text{ with } a_1 = p_1, \ldots, a_k = p_k)$$

where $u[p_i/a_i]$ is the process we get by substituting in u the atoms a_i by their codes p_i. This rule clearly states that the operation of atoms on an abstract object is that of their codes on the concrete representation of the object.

Let us sketch how this can be used to model CCS communication. Let sem be a given primitive object whose initial state is *free*, and which (exclusively) accepts the following operations:

$$\textit{free} \overset{P}{\mapsto} \textit{busy} \quad \text{and} \quad \textit{busy} \overset{V}{\mapsto} \textit{free}.$$

This is a kind of boolean semaphore, cf. [21]. We shall see that, as suggested in [33], a CCS *port* can be defined as the following *communication structure*:

$$\text{port} = ((\text{sem} \| \text{sem}') \text{ with } \textit{send} = (P; V'), \textit{receive} = (P'; V))$$

(with an obvious meaning for sem′, P' and V'). The mutual inclusion of *send* and *receive* will result from an indivisible interleaving of their codes, as in the sequence $P;P';V';V$ for instance. We shall also see how to define the CCS restriction $p\backslash\alpha$ in our calculus.

To sum up, our contribution relies on two main ideas: the first one is that it is worth putting some structure in what labels the arrows. The second is that it is worth setting up two kinds of arrows. One can think of CCS and related models as staying at the execution side, while functional languages stay at the operation side; what we attempt is to make up a unifying operational framework.

Note. One must regard our semantics of atomic actions as a preliminary proposal, formalizing the more concrete one of [7]. We do not prove any theoretical result about them. In particular our operation model is not yet clearly related to the system model of event structures.

Summary

In the first part we deal with our chosen system model, that of labelled event structures. We begin with a brief account on partial ordering approaches to concurrency. Then we introduce labelled event structures, together with a first syntax where the primitive constructs are sequential composition, parallel composition and sum.

Next we enrich this syntax with recursive definitions and show how to interpret terms as event structures. The last section of the first part contains a characterization of the class of event structures which are denoted by finite terms, and gives a complete axiomatization of the interpretation equality.

The second part of the paper is devoted to the execution model. We define our notion of transition on event structures, and show that this corresponds exactly to an operational semantics on terms. Then we discuss the associated strong bisimulation, which is called here equipollence, and give a complete axiomatization for it (for finite terms).

This third part presents the operation model, and our full calculus. We first introduce the notion of operation of programs on data. Then we give a syntax for data, which are called objects here. The syntax of terms is extended with constructs allowing processes to operate on objects. The transition relation of execution is extended to processes, and we introduce the transition relation on objects that formalizes the operation of processes. The last sections are concerned with abstraction and atomic actions. In particular, we show how to formulate some classical synchronization and communication primitives as atomic actions.

The reader will find in the three appendices a recapitulation of the technical material: the first one recollects the syntax, the second one contains the axioms and equational theories, and the third includes the rules of operational semantics.

PART I: THE SYSTEM MODEL

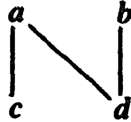
2. Syntax and interpretation

2.1. Algebras of posets: a survey

One of the first proposals in the area of partial order semantics of concurrency may be found in the work of Mazurkiewicz [40]. His formalization describes concurrency as an *independence* relation on actions, while causality is represented as sequencing of actions in a behaviour. This gives the notion of *trace* which is the equivalence class of a sequence up to commutation of independent actions (a more common meaning of the word "trace" is just that of a sequence, such as used by Hoare [32]). For instance, let $A=\{a, b, c, d\}$ be the set of actions and $I=\{(a, b), (b, c), (c, d)\}$ the independence relation; then the trace determined by the word *abcd* is the set

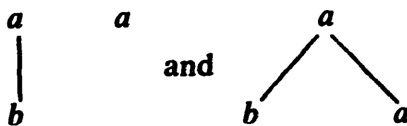
$$\{abcd, bacd, acbd, abdc, badc\}.$$

One can see that in the words of this set, b always precedes d, and a always precedes c and d. Thus one can represent this trace by a labelled poset N



(in this figure the order increases downwards). The partial ordering associated with a trace is the intersection of the linear orderings given by the words of the trace. This representation of traces by labelled posets—up to isomorphism—is given by Mazurkiewicz in [40, 41]. One may even characterize the labelled posets that represent traces, up to isomorphism, as shown by Shields [62] and Viennot [65]—another such characterization is given by Grabowski [27].

The set of traces on a set of actions A has an obvious algebraic structure: it inherits the structure of a monoid from the set A^* of sequences of actions, the operation being concatenation up to commutation of independent actions. Mazurkiewicz shows in [41] that such monoids satisfy some simplification properties, and Viennot [65] shows how to define directly the product of labelled posets representing traces. In fact, this kind of monoids, called *commutation monoids*, was introduced by Cartier and Foata in [11] to study combinatorial properties of rearrangements of sequences. Concurrency is implicit in the algebra of traces: there is only one operation to compose traces, namely that of concatenation. Let us now assume that we have two explicit constructs for programs: sequential and parallel composition. Then we can write, using a CCS-like syntax, a "program" $(ab|a)$, where a and b are actions. But there is no trace to represent its semantics. What we could do here is to interpret the parallel composition as the shuffle and thus get the set of behaviours $\{aab, aba\}$. However, as noted by Pratt [58], there is no longer uniqueness of labelled posets represented by this set; for example $\{aab, aba\}$ is the set of linearizations (or *sequentializations*) of both the labelled posets



The second one can be regarded as the interpretation of the term $a(b|a)$.

Labelled posets (up to isomorphism) seem to provide a sharp interpretation of concurrent programs. Let us give the definition: an A-labelled poset is a structure $(E, \leq, \lambda)$ where $\leq$ is an ordering on the set E of *events* and $\lambda : E \to A$ is the labelling. Informally an event $e \in E$ is an occurrence of the action $\lambda(e)$, and $e \leq e'$ means that e necessarily precedes e'. If two events e and e' are incomparable, we say that they are *concurrent*, and write $e \smile e'$. Structures of this kind have been introduced by Winkowski in [66, 67]; he imposes however a somewhat unfortunate restriction on his "partial sequences", namely that two concurrent events do not carry the same action. This rules out $(a|a)$ for instance.

As a matter of fact, labelled posets appear very often in the literature: they bear the name of "partial word" for Grabowski [27], of "pomset" (partially ordered

multiset) for Pratt and Gisher [24], of "A-poset" for Shields [62], and so on. Degano and Montanari propose in [17, 18] posets called "concurrent histories" in which they distinguish two kinds of labels (process types and actions). Petri net theory also uses a semantical object which is a special case of poset, that of "occurrence net", from which Reisig [59] extracts the "abstract processes" to formalize the computation of nets—the same notion was introduced by Grabowski [27]. In this paper we shall adopt Pratt's *pomset* terminology, cf. [58].

Assuming that pomsets are an appropriate model for concurrent programs, the question is: can they be supplied with an algebraic structure? In other words, we ask for a *syntax* to denote pomsets. An answer is given by Grabowski [27] who shows that every (finite) pomset may be built from atoms by means of sequential composition, renamings and a family of (multi-valued) operations called "sections"—these operations are similar to the parallel composition by intersection of TCSP [9] and to Milner's conjunction [46, 48]. However, this result essentially concerns posets, where the labelling is injective. Thus one has to assume an infinite set of actions A, which means that one cannot in general present the class of A-labelled posets as a free algebra generated by A, at least not with Grabowski's operations. The problem of finding a set of operators which are both operationally meaningful and sufficiently expressive does not yet have a satisfactory solution.

We shall content ourselves with the above mentioned constructs (parallel and sequential composition). Now the question is: which class of pomsets can we describe in this way? Here again the answer comes from Grabowski—and independently from Gischer [24]: this class is exactly that of *finite N-free pomsets*. We shall see later the precise definition of the N-freeness property. Roughly, an N-free pomset is one that does not contain the previously encountered pomset N. There exist some variations of this property: for example, one can consider pomsets or posets which do not strictly contain N, that is, do not contain an N-configuration for the covering relation. Grillet ([28], see also [55]) has shown that such posets are characterized by the property that each maximal chain intersects each maximal slice. An algebraic characterization of this class of posets has been given by Habib and Jegou [29]. They use a family of operations parameterized on a subset of maximal elements of the first argument, and on a subset of the minimal elements of the second one (Degano and Montanari introduce similar constructions on concurrent histories). Clearly, such operations, which depend on the names of events, do not provide a reasonable abstract syntax. Incidentally, let us note that the class of N-free pomsets, or more accurately of their line digraphs (cf. [36]) called *series parallel digraphs*, has been known for a long time by authors studying models of switching circuits [22, 60, 61]. These graphs also arise in problems of jobs scheduling (e.g.. topological sort), cf. [35].

To conclude we could point out that, for most of the models we mentioned, a process is a *set* of pomsets, which represent its possible computations; in other words a process is a language. This means that one may introduce a *sum* operator $p+q$ (nondeterministic choice) which is interpreted as set union. This entails the

linearity, or *distributivity* of sequential and parallel composition with respect to sum: $p(q+r)=pq+pr$ and $p|(q+r)=(p|q)+(p|r)$. Now it is well-known that models of this kind do not account for deadlock situations [44, 9]. For instance, if we assume a process $\mathbb{1}$ (similar to the NIL of CCS) such that $x+\mathbb{1}=x$, then we will have $ab=ab+a$ and $(a|b)=(a|b)+a$.

We shall not adopt such a linear interpretation of $p+q$: here the sum will be a true branching control structure. This construct is often used to model standard programming concepts such as (if ... then ... else ...) or (case ...) statements. For this reason we depart from pomsets as a system model, and rather adopt a framework in which the concepts of concurrency, sequentiality and nondeterminism take place at the same level. This framework is that of Winskel's *event structures* [68].

2.2. Labelled event structures and terms

For some technical reasons that will become clear later, our definition of event structures is a slight variation of Winskel's one. Moreover, in this first part we only deal with event structures in themselves, not with the domain of configurations they determine. Configurations will be introduced in the next part. At some points we shall assume knowledge of the work of Nielsen, Plotkin, and Winskel [49] which shows how to derive (labelled) event structures from some kind of (labelled)Petri nets; thus we shall feel free to use standard concepts of net theory (cf. [23]) when dealing with such derived event structures.

As usual $\{0,1\}^*$ denotes the set of words over the alphabet $\{0,1\}$; the concatenation of two words u and v is uv, the empty word is ε.

Definition 2.1. Let A be a nonempty set. An *A-labelled event structure* (*A-LES* for short) is a structure $(E, \leq, \#, \lambda)$ where

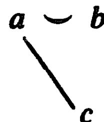
(i) $E \subseteq \{0,1\}^*$ is the set of events,
(ii) $\leq$ is a partial order on E, the causality relation,
(iii) $\# \subseteq E\times E-(\leq\cup\geq)$ is the symmetric conflict relation,
(iv) $\lambda : E\to A$ is the labelling function.

In what follows we let $a, b, c, \ldots$ range over the set A of *atoms*. Two events e, e' in E are *concurrent* if they are neither comparable nor in conflict, that is, $e \smile e'$ where

$$\smile \overset{\text{def}}{=} E\times E-(\leq\cup\geq\cup\#).$$

This is a symmetric irreflexive relation. Note that, by definition, the relations $\leq\cup\geq$, $\#$, and $\smile$ set a partition upon $E\times E$.

We shall always draw structures up to isomorphism, that is, omitting the name of events; in the figures the order $\leq$ increases downwards and only one of the remaining relations ($\#$ or $\smile$) is explicitly shown. For instance, ∇



is a structure with three events, e, e' and e'', respectively labelled a, b and c such that e is a cause of e'', e and e' are concurrent and e' and e'' are in conflict. This event structure is derived from the net shown in Fig. 1 which is the typical example of *asymmetric confusion* (cf. [23]). Note that we do not require Winskel's axioms of *conflict heredity*, stating

$$e''\#e \leqslant e' \Rightarrow e''\#e'.$$

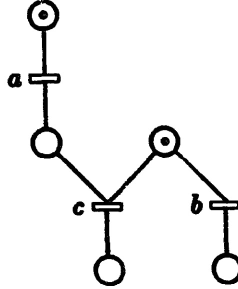
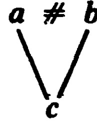


Fig. 1.

Therefore, we do not rule out structures such as, for instance,



We use $\mathscr{L}(A)^\infty$ for the set of A-labelled event structures and $\mathscr{L}(A)$ for the set of finite ones. These sets are naturally supplied with an algebraic structure: let V be one of $\leqslant$, $\smile$, $\#$ and S_0, S_1 be A-LES's; then $S_0\,(V)\,S_1$ is the structure we get by juxtaposing S_0 and S_1 and setting the V relation between the events of S_0 and S_1. When V is $\leqslant$, this is called *sequential composition* of S_0 and S_1 and denoted $S_0;S_1$, whereas if V is $\smile$, this is the *parallel composition* $S_0\|S_1$ and in the case $V=\#$, this is the sum S_0+S_1. The formal definition is the following: assuming

$$S_i=(E_i, \leqslant_i, \#_i, \lambda_i) \quad \text{for } i\in\{0,1\}$$

one defines $S_0\,(V)\,S_1$ to be $(E, \leqslant, \#, \lambda)$, where

- $E=E_0 \dot{\cup} E_1$, i.e., $E=\{0u \mid u\in E_0\}\cup\{1u \mid u\in E_1\}$;
- $ix\leqslant jy \Leftrightarrow i=j$ and $x\leqslant_i y$ or $V=\leqslant$, $i=0$ and $j=1$;
- $ix\#ju \Leftrightarrow i=j$ and $x\#_i y$ or $V=\#$ and $i\neq j$;
- $\lambda(ix)=\lambda_i(x)$.

These operations are naturally defined up to isomorphism. That is, denoting by $P\rightleftharpoons Q$ the relation "P and Q are isomorphic", we have

$$P\rightleftharpoons P' \text{ and } Q\rightleftharpoons Q' \Rightarrow \begin{cases} P;Q\rightleftharpoons P';Q' \\ P+Q\rightleftharpoons P'+Q', \\ P\|Q\rightleftharpoons P'\|Q'. \end{cases}$$

Thus $\mathscr{L}(A)^\infty/\rightleftharpoons$ and $\mathscr{L}(A)/\rightleftharpoons$ inherit the algebraic structure.

Now we have a *syntax* to denote finite A-LES's; we shall deal with infinite ones in a next section. This abstract syntax is the set $\mathbb{T}(A)$ of terms built according to the following formation rules.

Finite terms

(i) $\mathbb{1}$ is a term of $\mathbb{T}(A)$ and every atom $a \in A$ is a term of $\mathbb{T}(A)$;
(ii) if p and q are terms of $\mathbb{T}(A)$ then so are $(p;q)$, $(p\|q)$ and $(p+q)$.

Let $\mathcal{S}(p)$ be the labelled event structure denoted by the term p, defined as follows:

$$\mathcal{S}(\mathbb{1}) = (\emptyset, \emptyset, \emptyset, \emptyset) \qquad \text{(the empty structure)},$$

$$\mathcal{S}(a) = (\{\varepsilon\}, =, \emptyset, \lambda) \quad \text{with } \lambda(\varepsilon) = a,$$

$$\mathcal{S}(p;q) = \mathcal{S}(p);\mathcal{S}(q),$$

$$\mathcal{S}(p\|q) = \mathcal{S}(p)\|\mathcal{S}(q),$$

$$\mathcal{S}(p+q) = \mathcal{S}(p)+\mathcal{S}(q).$$

The symbol $\mathbb{1}$ will be used also for the empty structure and its isomorphism class. Let us see a few examples: the term $(a+b);(c\|d)$ denotes the structure

This and the simpler term $(a+b);c$ show why we cannot assume the axiom of conflict heredity. The term $(a\|b)+c$ is interpreted as

$$a\#c\#b$$

(where $a \smile b$, and there is no nontrivial causal dependency) and it is an example of "symmetric confusion" (see [23, 49]), derived from the net shown in Fig. 2.

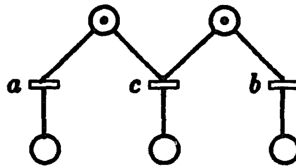


Fig. 2.

It will be convenient to denote $[\![S]\!]_{\cong}$, or more simply $[\![S]\!]$, the isomorphism class of an A-LES S. Then the *interpretation* of a term $p \in \mathbb{T}(A)$ is:

$$\mathcal{I}(p) \stackrel{\text{def}}{=} [\![\mathcal{S}(p)]\!].$$

In a next section we shall give a characterization of the set of finite structures which

are interpretations of terms up to isomorphism, that is,

$$\mathcal{T}(A) \stackrel{\text{def}}{=} \mathcal{I}(\mathbf{T}(A)) \subseteq \mathcal{L}(A)/\rightleftharpoons.$$

Moreover, we shall give an axiomatization of the *interpretation equality*:

$$p =_{\mathcal{I}} q \stackrel{\text{def}}{\Leftrightarrow} \mathcal{I}(p) = \mathcal{I}(q) \Leftrightarrow \mathcal{I}(p) \rightleftharpoons \mathcal{I}(q).$$

2.3. *Infinite structures and recursive definitions*

In this section we enrich the syntax of terms with recursive definitions in order to be able to denote some infinite A-LES's. The syntactical apparatus of recursive definitions, and their semantical interpretation in ordered domains are nowadays standard matters of the theory of programming language semantics, with which we assume the reader to be familiar.

Winskel introduces in [70] an approximation ordering between LES's which yields a cpo structure. This ordering is simply the "substructure" or *restriction* ordering, defined as follows:

$$S' \subseteq S \stackrel{\text{def}}{\Leftrightarrow} \begin{cases} S = (E, \leqslant, \#, \lambda) \text{ and } \exists F \subseteq E, \\ S' = S\lceil F = (F, \leqslant \cap (F \times F), \# \cap (F \times F), \lambda \lceil F). \end{cases}$$

Since this relation is defined by means of set inclusion it enjoys some pleasant properties. To state them let us recall some standard order-theoretic notions:

(i) In a poset $(X, \leqslant)$ a subset C of X is *locally bounded* (or *consistent*) if every finite subset of C has an upper bound (in X).

(ii) The poset $(X, \leqslant)$ is *consistently complete* if every locally bounded subset C has a lub $\bigsqcup C$.

(iii) Let $(X, \leqslant)$ be consistently complete. A point $x \in X$ is called *finite* if for every directed subset Z such that $x \leqslant \bigsqcup Z$ there exists a $z \in Z$ such that $x \leqslant z$.

(iv) A consistently complete poset is *algebraic* if each of its points is the lub of the finite points it dominates.

We assume the notion of *continuous* function over consistently complete posets to be well-known. The following fact is not a surprise.

Proposition 2.2. *The poset* $(\mathcal{L}(A)^{\infty}, \subseteq)$ *is a consistently complete algebraic poset whose set of finite points is* $\mathcal{L}(A)$. *Moreover, the operations of sequential composition, sum, and parallel composition are continuous over this poset.*

The straightforward proof is omitted. The relevant consequence of this fact is that we can solve *systems of algebraic equations* in the poset $(\mathcal{L}(A)^{\infty}, \subseteq)$. More accurately, this result allows us to interpret terms containing recursive definitions, which could take the following syntactic form:

$$(\text{def rec } x_1 = p_1, \ldots, x_n = p_n \text{ in } q).$$

In order to extend the syntax with such a construct, we assume a denumerable set X of *identifiers*, disjoint from the set A of atoms. As usual the (def rec ... in ...) construct *binds* the defined identifiers, and thus introduces notions of free and bound occurrences of identifiers. We define for each finite subset Y of X the set $\mathbf{T}^{\mathrm{rec}}(A \cup Y)$ of terms whose free identifiers are those of Y—so that the set of *closed* terms is $\mathbf{T}^{\mathrm{rec}}(A)$. We shall use a restricted syntax for recursive definitions, allowing only definitions of the form (def $x = p$ in x), what we denote $\mu x.p$ (for some authors this is rec $x.p$ or fix $x.p$, cf. [46]). The set of "finite" terms, built without recursive definitions but with free identifiers in Y is $\mathbf{T}(A \cup Y)$. We shall restrict the formation of sequential composition $(p;q)$ to the case where p is closed and finite—this is to avoid problems with termination. Then the formation rules for (recursive) terms are as follows.

Terms

(i) $\mathbb{1}$ and every atom $a \in A$ are terms of $\mathbf{T}^{\mathrm{rec}}(A)$;
(ii) if p and q are terms of $\mathbf{T}(A)$ and $\mathbf{T}^{\mathrm{rec}}(A \cup Y)$ respectively, then $(p;q)$ is a term of $\mathbf{T}^{\mathrm{rec}}(A \cup Y)$;
(iii) if p and q are terms of $\mathbf{T}^{\mathrm{rec}}(A \cup Y)$ and $\mathbf{T}^{\mathrm{rec}}(A \cup Y')$ respectively, then $(p\|q)$ and $(p+q)$ are terms of $\mathbf{T}^{\mathrm{rec}}(A \cup Y \cup Y')$;
(iv) an identifier $x \in X$ is a term of $\mathbf{T}^{\mathrm{rec}}(A \cup \{x\})$;
(v) if x is an identifier and p is a term of $\mathbf{T}^{\mathrm{rec}}(A \cup Y)$, then $\mu x.p$ is a term of $\mathbf{T}^{\mathrm{rec}}(A \cup Y - \{x\})$.

We obviously have $\mathbf{T}(A) \subseteq \mathbf{T}^{\mathrm{rec}}(A)$. Since recursive definitions bind identifiers, we shall regard as syntactically identical those terms which only differ in the names of bound identifiers. For instance,

$$\mu x.(b;(x\|y)) = \mu x'.(b;(x'\|y)).$$

We shall use $q[p/x]$, to denote the term that we get by substituting p for the identifier x in all its free occurrences in q; bound identifiers of q may have to be renamed to avoid captures of free identifiers of p. For instance,

$$(\mu x.(b;(x\|y)))[(a;x)/y] = \mu x'.(b;(x'\|(a;x)).$$

In order to interpret closed terms into $\mathscr{L}(A)^{\infty}$—or more accurately into $\mathscr{L}(A)^{\infty}/\rightleftharpoons$— we need the notions of *unfolding* and *immediate approximation* of a term. A term p is an unfolding of another term q if we get p from q by substituting for some identifiers their (recursive) definition. Then the sets $\mathscr{U}(p)$ of unfoldings of terms p of $\mathbf{T}^{\mathrm{rec}}(A)$ are the least ones satisfying the following clauses:

(i) $p \in \mathscr{U}(p)$ for any term p;
(ii) if $p' \in \mathscr{U}(p)$ and $q' \in \mathscr{U}(q)$, then $(p';q') \in \mathscr{U}(p;q)$, $(p'\|q') \in \mathscr{U}(p\|q)$ and $(p'+q') \in \mathscr{U}(p+q)$;
(iii) if $q \in \mathscr{U}(p[\mu x.p/x])$, then $q \in \mathscr{U}(\mu x.p)$.

For instance, let

$$r = \mu x.(a;(b\|x)).$$

Then

$$\mathcal{U}(r) = \{r_n \mid 0 \le n\} \quad \text{where} \begin{cases} r_0 = r, \\ r_{n+1} = (a;(b\|r_n)). \end{cases}$$

The unfolding process is *confluent*, that is,

$$p' \in \mathcal{U}(p) \text{ and } p'' \in \mathcal{U}(p) \Rightarrow \mathcal{U}(p') \cap \mathcal{U}(p'') \neq \emptyset.$$

This is a standard fact. Note that if r is a closed term, then all its unfoldings are also closed. The immediate approximation $\varpi(p)$ of a closed term p is what we get by simply forgetting the recursive definitions, that is, by substituting $\mathbb{1}$ for all the subterms $\mu x.p$:

(i) $\varpi(\mathbb{1}) = \mathbb{1}$;
(ii) $\varpi(a) = a$ for all $a \in A$;
(iii) $\varpi(p;q) = (\varpi(p);\varpi(q))$, $\varpi(p\|q) = (\varpi(p)\|(q))$ and $\varpi(p+q) = (\varpi(p)+\varpi(q))$;
(iv) $\varpi(\mu x.p) = \mathbb{1}$.

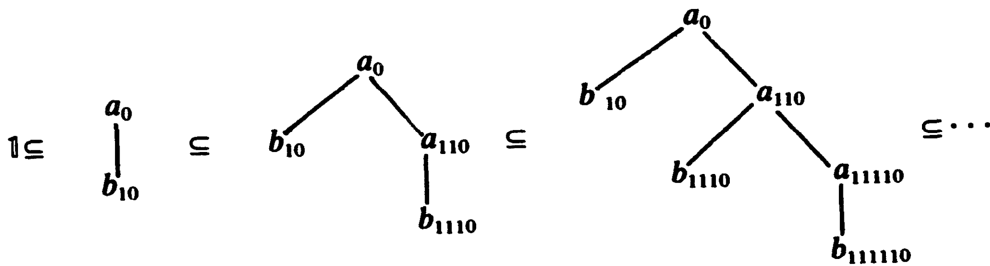
For instance, the set $\varpi(\mathcal{U}(r))$ associated with the previous r is the sequence $\varpi(r_n)$

$$\varpi(r_0) = \mathbb{1}, \qquad \varpi(r_{n+1}) = (a;(b\|\varpi(r_n))).$$

Note that $\varpi(p)$ is always a "finite" term, belonging to $\mathbf{T}(A)$. Another standard fact is that unfolding increases the immediate approximation, that is,

$$\forall p \in \mathbf{T}^{\text{rec}}(A) \quad p' \in \mathcal{U}(p) \Rightarrow \mathcal{J}(\varpi(p)) \subseteq \mathcal{J}(\varpi(p')).$$

Continuing the above example we get for the following sequence $\mathcal{J}(\varpi(r_n))$:



Here the names of the events are shown as indices of the labels to make evident the ordering relation.

A consequence of the previous facts is that, for any closed term p, the set

$$\{\mathcal{J}(\varpi(p')) \mid p' \in \mathcal{U}(p)\}$$

of (finite) *approximants* of p is pairwise consistent (and, in fact, directed), and thus has a lub in $\mathcal{L}(A)^\infty$. We are then able to define the interpretation of terms of $\mathbf{T}^{\text{rec}}(A)$:

$$\mathcal{J}^\infty(p) \stackrel{\text{def}}{=} \bigsqcup \{\mathcal{J}(\varpi(p')) \mid p' \in \mathcal{U}(p)\}.$$

As usual, this interpretation could be defined as the limit of an increasing sequence if we let:

(i) $\kappa(\mathbb{1})=\mathbb{1}$;

(ii) $\kappa(a)=a$ for all $a\in A$;

(iii) $\kappa(p;q)=(\kappa(p);\kappa(q))$, $\kappa(p\|q)=(\kappa(p)\|\kappa(q))$ and $\kappa(p+q)=(\kappa(p)+\kappa(q))$;

(iv) $\kappa(\mu x.p)=p[\mu x.p/x]$.

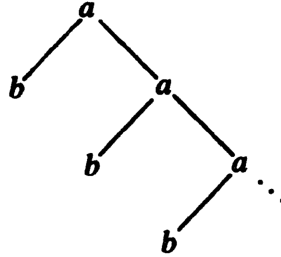
Then $\mathcal{J}^\infty(p)=\bigsqcup\{\mathcal{J}(\varpi(\kappa^n(p)))\mid n\geqslant 0\}$. It should be clear that this definition extends that of $\mathcal{J}$, that is, $\mathcal{J}^\infty(p)=\mathcal{J}(p)$ for $p\in\mathbf{T}(A)$. Just as for "finite" terms, we define

$$\mathcal{I}(p)\overset{\text{def}}{=}[\![\mathcal{J}^\infty(p)]\!] \quad\text{and}\quad \mathcal{T}^\infty(A)\overset{\text{def}}{=}\mathcal{I}(\mathbf{T}^{\text{rec}}(A))\subseteq\mathcal{L}(A)^\infty/\rightleftharpoons.$$

Then the interpretation equality is

$$p=_{\mathcal{I}}q\overset{\text{def}}{\Leftrightarrow}\mathcal{I}(p)=\mathcal{I}(q)\Leftrightarrow\mathcal{J}^\infty(p)\rightleftharpoons\mathcal{J}^\infty(q).$$

Pursuing the above example, one can see that $\mathcal{I}(\mu x.(a;(b\|x)))$ is



where there is no conflict.

3. Characterization

One may remark that in $\mathcal{L}(A)^\infty/\rightleftharpoons$ the three operations previously defined are associated and have $\mathbb{1}$ as neutral element; moreover, the sum and parallel composition are commutative. This suggests the following definition.

Definition 3.1. A *trioid* is an algebra $(T,;,\|,+,1)$ satisfying the axioms

(i) $(T,;,1)$ is a monoid:

A0: $(p;(q;r))=((p;q);r)$,

U0: $(p;1)=p=(1;p)$;

(ii) $(T,\|,1)$ is a commutative monoid:

A1: $(p\|(q\|r))=((p\|q)\|r)$,

U1: $(p\|1)=p=(1\|p)$,

C1: $(p\|q)=(q\|p)$;

(iii) $(T,+,1)$ is a commutative monoid:

A2: $(p+(q+r))=((p+q)+r)$,

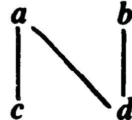
U2: $(p+1)=p=(1+p)$,

C2: $(p+q)=(q+p)$.

Let Θ be the equational theory whose axioms are A0 to A2, U0 to U2, C1 and C2, and let $=_\Theta$ be the congruence on $\mathrm{T}^{\mathrm{rec}}(A)$ generated by these equations. Then we have an obvious *soundness* property:

$$p =_\Theta q \Rightarrow p =_{\mathscr{I}} q.$$

We now wish to check whether or not a converse *completeness* property holds *for "finite" terms*, that is, terms of $\mathrm{T}(A)$. First we shall see that not all finite labelled event structures are interpretations of terms. We have already mentioned the fact that the structure N



(without conflict) is known to be typical of those that cannot be expressed by means of sequential and parallel composition, cf. [24, 27]. To obtain completeness, we thus want to find a class of A-LES's which do not contain N. In order to define this class and state our characterization result we need to introduce some notations. Let $R \subseteq E \times E$ be a relation on a set E.

(i) $R^\sigma = R \cup R^{-1}$ is the symmetric closure of R.

(ii) $R^\varepsilon = R^\sigma \cup R^0$ is the reflexive and symmetric closure of R, which we shall call the *R-comparability* relation.

(iii) $\ddagger(R) = (E \times E) - R^\varepsilon$ is the symmetric, irreflexive R-incomparability relation.

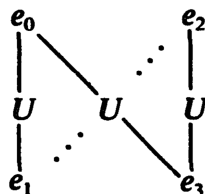
(iv) $\sim_R = (R^\sigma)^*$ is the equivalence generated by R whose classes are the connected components with respect to the R-comparability relation.

For instance, the comparability relations determined by $\#$ and $\smile$ are simply their reflexive closures, whereas the $\leqslant$-comparability is $\leqslant \cup \geqslant$ which we denote by $\diamond$, and $\ddagger(\diamond) = \# \cup \smile$. In order to avoid many useless repetitions we shall name each of the relations $\leqslant$, $\#$, $\smile$ a *connective* of a given structure S.

The first property we shall require is N-*freeness*; an A-LES S is N-free if it satisfies

$$\text{N-}\textit{freeness}\quad \begin{cases} \text{for } U \text{ a connective of } S. \\ \text{if } e_0\, U\, e_1 \text{ and } e_0 \ddagger(U)\, e_2 \\ \text{if } e_2\, U\, e_3 \text{ and } e_1 \ddagger(U)\, e_3, \\ \text{then } e_0\, U\, e_3 \Rightarrow e_2\, U\, e_1. \end{cases}$$

This property, which is obviously preserved by isomorphism, may be drawn



This typically precludes a structure such as $a\#b\#c\#d$ (where $a \smile c$, $b \smile d$, and $a \smile d$) which is derived (see [49]) from the Petri net shown in Fig. 3.

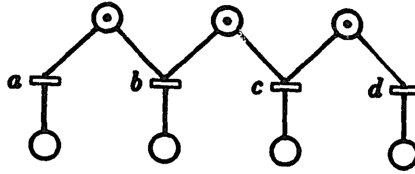


Fig. 3.

N-freeness is also related to Petri's notion of *K-density*, or more accurately *N-density* [52], see [28, 55].

N-freeness is not enough by itself to characterize the class of A-LES's denoted by terms. Here we need another requirement which we may call the *triangle-freeness property*: a structure S satisfies this property (referred to as the ∇-freeness property) if it does not contain a configuration

$$\textit{triangle} \quad e \diamond e' \# e'' \smile e.$$

This precludes the typical situation of "asymmetric confusion" (cf. [23, 49]), that we have already seen above.

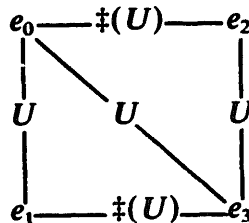
In fact the "behavioural" properties of N-freeness and ∇-freeness may be combined in a single one—which is somehow more natural when looking for a property preserved by the operations.

Lemma 3.2. *An A-labelled event structure S satisfies the* N*-freeness and* ∇*-freeness properties if and only if it satisfies the property*

$$\mathbf{X} \quad \begin{cases} \text{for } U \text{ and } V \text{ among } \leqslant, \#, \smile \text{ with } U \neq V \\ \text{if } e_0\, U\, e_1 \text{ and } e_0 \ddagger(U)\, e_2 \\ \text{if } e_2\, U\, e_3 \text{ and } e_1 \ddagger(U)\, e_3, \\ \text{then } e_0\, V\, e_3 \Rightarrow \{e_0, e_1\} \times \{e_2, e_3\} \subseteq V. \end{cases}$$

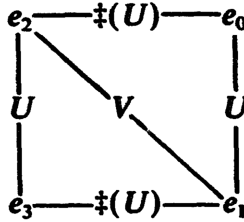
Proof. It is easy to see that if S satisfies the X-property, then it is ∇-free since a triangle $e \diamond e' \# e'' \smile e$ does not satisfy the X-property (if we let $e_0 = e$, $e_1 = e'$, or $e_0 = e'$, $e_1 = e$, and $e_2 = e'' = e_3$).

Let us prove that the X-property also implies N-freeness: let us assume that S satisfies X and contains a configuration



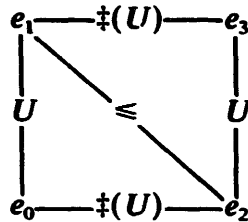
Regarding the relationship between e_1 and e_2, the a priori possible cases are the following:

(1) if $e_2\ V e_1$, with $V \in \{\leq, \#, \smile\}$, either $V = U$ and we are done, or $V \neq U$. Then we apply the X-property to the following configuration:



We get $e_3\ V e_0$, together with the hypothesis $e_0\ U e_3$. Since U and V are two distinct connectives, at least one of them is symmetric; therefore, one has either $e_3\ U e_0$, which contradicts $e_3\ V e_0$, or $e_0\ V e_3$, which contradicts $e_0\ U e_3$.

(2) The only remaining case is $e_1 \leq e_2$; we cannot have $U = \leq$, otherwise we would have $e_0 \leq e_1 \leq e_2$, contradicting $e_0 \ddagger(U)\ e_2$. Then $U^{-1} = U$ and we can apply the X-property to the configuration



We get $e_0 \leq e_3$, but this contradicts $e_0\ U e_3$ since U is a connective distinct from $\leq$. Therefore it must be the case that $e_2\ U e_1$, and we have shown that the X-property implies N-freeness.

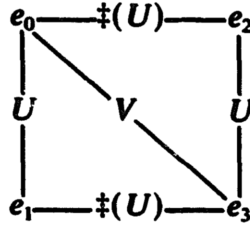
In order to prove that an N-free and ∇-free LES satisfies the X-property, we shall use the following claim.

Claim. *If a structure S is* N*-free, then it satisfies*

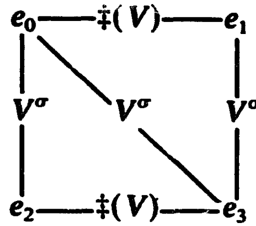
$$\text{N}' \begin{cases} \textit{for } U \textit{ a connective of } S \\ \textit{if } e_0\ U^\sigma\ e_1 \textit{ and } e_0 \ddagger(U)\ e_2 \\ \textit{if } e_2\ U^\sigma\ e_3 \textit{ and } e_1 \ddagger(U)\ e_3, \\ \textit{then } e_0\ U^\sigma\ e_3 \Rightarrow \{e_0, e_2\} \times \{e_1, e_3\} \subseteq U \textit{ or } \{e_0, e_2\} \times \{e_1, e_3\} \subseteq U^{-1}. \end{cases}$$

Proof. Let us assume the hypothesis of N′ with $U = \leq$ (the claim is trivial when $U = \#$ or $U = \smile$ since these connectives are symmetric). It cannot be the case that $e_0 \leq e_3 \leq e_2$ since $e_0 \ddagger(U)\ e_2$; similarly, $e_2 \leq e_3 \leq e_0$ is precluded. Since $e_1 \ddagger(U)\ e_3$, we cannot have $e_3 \leq e_0 \leq e_1$ nor $e_1 \leq e_0 \leq e_3$. Therefore, the only possible cases are $e_0 \leq e_1$. and $e_2 \leq e_3$ and $e_0 \leq e_3$; that is, the hypothesis of N, whence $e_2 \leq e_1$, or $e_0 \geq e_1$, and $e_2 \geq e_3$ and $e_0 \geq e_3$ where, rotating the configuration, we conclude $e_2 \geq e_1$ by the N-property. □

Proof of Lemma 3.2 (*continued*). Let us assume that S is ∇-free, satisfies N′, and contains the following configuration:



with $U, V \in \{\leq, \#, \smile\}$ and $U \neq V$ (note that $e_0 \neq e_2$ and $e_1 \neq e_3$). Then we have $e_0\ W^\sigma\ e_2$ for some connective W; it cannot be the case that $W = U$ since $e_0 \ddagger(U)\ e_2$. If $e_2 = e_3$, then $W = V$; otherwise since S is ∇-free, we must have $W^\sigma = V^\sigma$. In any case $e_0\ V^\sigma\ e_2$, and similarly, $e_1\ V^\sigma\ e_3$. We then have the configuration



Since the hypothesis is $e_0\ V e_3$, we conclude by the N′-property that $\{e_0, e_1\} \times \{e_2, e_3\} \subseteq V$. □

We can finally define the intended class of structures as follows.

Definition 3.3. The set $\mathscr{X}(A)^\infty$ (respectively $\mathscr{X}(A)$) is the set of A-LES's (respectively finite A-LES's) satisfying the X-property.

The set of structures $\mathscr{X}(A)^\infty$ (or, more accurately, $\mathscr{X}(A)^\infty/\rightleftharpoons$) is a generalization of Grabowski-Gischer's class of *N-free pomsets* [27, 24]. Clearly, the X-property is *hereditary*; this means that if $S' \subseteq S$ & $S \in \mathscr{X}(A)^\infty$, then $S' \in \mathscr{X}(A)^\infty$. We can now state the announced result, which generalizes Grabowski-Gischer's one.

Theorem 3.4 (Characterization). *The structure* $(\mathscr{X}(A)/\rightleftharpoons, ;, \|, +, \mathbb{1})$ *is the free trioid generated by A. In particular,*

(i) $S \in \mathscr{X}(A) \Leftrightarrow \exists p \in \mathrm{T}(A)\ \mathscr{S}(p) \rightleftharpoons S$; *therefore,* $\mathscr{T}(A) = \mathscr{X}(A)/\rightleftharpoons$;

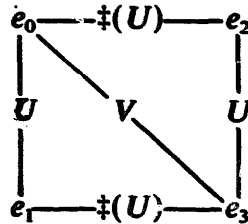
(ii) $p =_{\mathscr{S}} q \Leftrightarrow p =_\Theta q$.

The complete proof is rather long, involving some straightforward parts, which are omitted. One has to prove that $\mathscr{X}(A)/\rightleftharpoons$ is a trioid isomorphic to $\mathrm{T}(A)/=_\Theta$. We have already seen that the algebra $\mathscr{L}(A)/\rightleftharpoons$ is a model of the theory Θ. Thus the first thing to see is that the operations preserve the X-property; an immediate

consequence will be that $\mathscr{X}(A)$ is a trioid which contains the denotation $\mathscr{J}(p)$ of every ("finite") term $p \in \mathbf{T}(A)$, and then $\mathscr{T}(A) \subseteq \mathscr{X}(A)/\rightleftharpoons$.

Lemma 3.5. *If $S_0, S_1 \in \mathscr{X}(A)^\infty$, then $S_0;S_1$, S_0+S_1 and $S_0\|S_1$ are in $\mathscr{X}(A)^\infty$.*

Proof. Let $S_i = (E_i, \leqslant_i, \#_i, \lambda_i)$, let $F_i = \{iu \mid u \in E_i\}$ for $i = 0, 1$, and let $\{e_0, e_1, e_2, e_3\}$ be events of $S_0\,(W)\,S_1$ which satisfy the hypothesis of the X-property, that is:



with $U, V \in \{\leqslant, \#, \smile\}$ and $U \neq V$. The proof that we then have the desired conclusion

$$\{e_0, e_1\} \times \{e_2, e_3\} \subseteq V$$

proceeds by case inspection on the respective position of the events e_0, e_1, e_2, e_3.

(1) If they are all in the same F_i, then we are done.

(2) It is impossible that an F_i contains exactly one of these events: for instance, assume that $e_0 \in F_0$ and $\{e_1, e_2, e_3\} \subseteq F_1$; by definition of $S_0\,(W)\,S_1$, we have $F_0 \times F_1 \subseteq W$, therefore $e_0\,W e_1$, whence $W = U$, and $e_0\,W e_2$ but this contradicts the hypothesis $e_0 \ddagger(U)\, e_2$. All the other cases are similar.

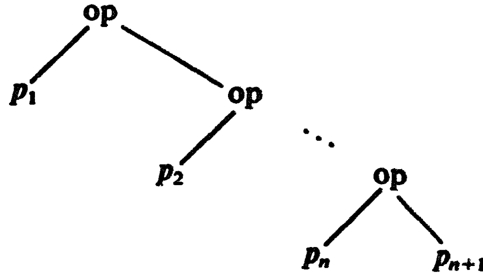
(3) The same argument shows that it is impossible that one of F_0, F_1 contains $\{e_0, e_3\}$ while the other contains $\{e_1, e_2\}$. On the other hand, if, for instance, $\{e_0, e_2\} \subseteq F_0$ and $\{e_1, e_3\} \subseteq F_1$, then we would have $e_0\,W e_1$ and $e_0\,W e_3$, hence $U = W = V$, but this contradicts $U \neq V$. The only remaining case is $\{e_0, e_1\} \subseteq F_0$ and $\{e_2, e_3\} \subseteq F_1$ (or possibly the converse if V is $\#$ or $\smile$), whence $W = V$ and $\{e_0, e_1\} \times \{e_2, e_3\} \subseteq V$. $\square$

Next one has to show that each element of $\mathscr{X}(A)/\rightleftharpoons$ is the interpretation of a term of $\mathbf{T}(A)$, uniquely up to $=_\Theta$. As usual, this completeness property rests upon the existence of *normal forms* for terms. These can be described as follows: let $\mathscr{N}(A) = \{\mathbb{1}\} \cup \mathscr{W}(A)$ where $\mathscr{W}(A)$ is the least set of terms built according to the following rules:

(i) every atom $a \in A$ is in $\mathscr{W}(A)$ and has no *head operator*,

(ii) if $p \in \mathscr{W}(A)$ does not have ; (respectively, $\|$, $+$) as head operator and if $q \in \mathscr{W}(A)$, then $(p; q)$ (respectively $(p\|q)$, $(p+q)$) is in $\mathscr{W}(A)$ and has ; (respectively $\|$, $+$) as head operator.

One gets normal forms by cancelling the unit and using associativity to shift arguments to the right. Therefore, a typical normal form (if it is not $\mathbb{1}$ nor an atom) may be drawn as



where the head operator op is either ; or $\|$ or $+$, and $p_1, p_2, \ldots, p_n, p_{n+1}$ do not have op as head operator.

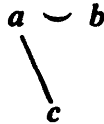
Proposition 3.6. *Let Γ be the theory whose axioms are* A0 *to* A2 *and* U0 *to* U2, *and* Υ *be the theory consisting of* A0 *to* A2, C1 *and* C2 (*cf.* Appendix B). *Then*

(i) *for each term $p \in \mathbf{T}(A)$ there exists a normal form $t \in \mathcal{N}(A)$ such that $p =_\Gamma t$;*

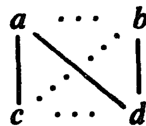
(ii) *for two normal forms $t, t' \in \mathcal{N}(A)$, $t =_\Theta t' \Leftrightarrow t =_\Upsilon t'$.*

This is a standard result. The proof is omitted.

The crux of the characterization theorem's proof is the following property: for every finite nonempty nonatomic labelled event structure satisfying the X-property, the set of events is connected for exactly one of the connectives $\leqslant$, $\smile$, $\#$ (in fact, this is a purely graph-theoretical result); this relation gives the head operator of the term which denotes the structure. The existence of such a connective comes from the ∇-freeness property, whereas uniqueness comes from N-freeness (or, more accurately, from N′). It is clear that ∇



is not connected for any of the connectives, and one cannot find a head operator for a term which would denote it. On the other hand, the structure N



(where the dotted lines stand either for $\#$ or for $\smile$) is connected for two connectives; here again one cannot choose a head operator for a term which would denote it.

Lemma 3.7. *Let $S=(E, \leqslant, \#, \lambda)$ be an A-LES in $\mathscr{X}(A)$.*

(i) *there exists a connective U of S for which E is connected, that is, $\#(E/\sim_U)=1$;*

(ii) *moreover, if $\#(E)>1$, then E is not connected for the U-incomparability relation $\ddagger(U)$, and thus is not connected for any of the other connectives.*

Proof. We first show that there is one such relation U, for each $S \in \mathscr{X}(A)$. Suppose not, and let C be a *maximal* (w.r.t. inclusion) subset of E connected for some connective. From our assumption, $C \neq E$, so let $e \in E - C$. Then e is connected in the same way ($\diamond$, $\#$ or $\smile$) with all the elements of C, otherwise S would contain a triangle. But then $\{e\} \cup C$ is, for some connective, a connected subset of E which strictly contains C.

Now, to prove the second point, let us assume that E is connected for both U and $\ddagger(U)$ for U among $\diamond$ (since $E \leqslant$-connected$\Leftrightarrow\diamond$-connected), $\#$ and $\smile$. Let F be a *minimal* (w.r.t. inclusion) subset of E which is both U- and $\ddagger(U)$-connected and such that $\#(F) > 1$. Then $\#(F) > 2$ since one cannot build a two-element structure which is connected for two exclusive relations. So let $e_3 \in F$; since $F - \{e_3\}$ is not connected for both U and $\ddagger(U)$, let us assume, for instance, that $F - \{e_3\}$ is not connected for U, that is,

$$(F - \{e_3\})/\sim_U = \{F_1, \ldots, F_m\} \quad \text{with } m > 1.$$

Then

$$\exists i (1 \leqslant i \leqslant m)\, \exists e \in F_i \quad e_3\, \ddagger(U)\, e;$$

otherwise F could not be $\ddagger(U)$-connected. Similarly,

$$\forall i (1 \leqslant i \leqslant m)\, \exists e \in F_i \quad e_3 U e.$$

So let G be an F_i such that $\exists e \in F_i\ e_3\, \ddagger(U)\, e$ and H be $\bigcup_{i \neq j} F_j$. Let $e \in G$ and $e' \in G$ be such that $e_3\, \ddagger(U)\, e$ and $e_3\, U e'$. Since G is U-connected, there exists a sequence of events of G which U-connects e and e'; an easy induction on the length of such a sequence shows that one has

$$\exists e_0 \in G\, \exists e_1 \in G \quad e_3\, U e_0 \text{ and } e_0\, U e_1 \text{ and } e_1\, \ddagger(U)\, e_3.$$

If we choose an $e_2 \in H$ such that $e_3\, U e_2$, we may figure the situation as shown in Fig. 4. By definition of G and H, $e_0\, \ddagger(U)\, e_2$ and $e_1\, \ddagger(U)\, e_2$, but this contradicts the N′-property, which is a consequence of the X-property.

The proof is the same when $F - \{e_3\}$ is not $\ddagger(U)$-connected. $\square$

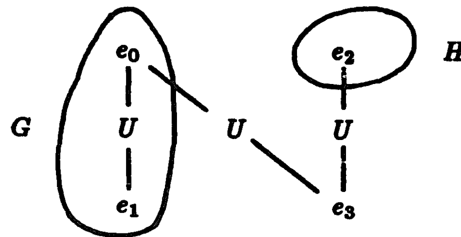


Fig. 4

Proof of Theorem 3.4. We can now prove

$$\forall S \in \mathscr{X}(A)\, \exists t \in \mathscr{N}(A) \quad \mathscr{S}(t) \rightleftharpoons S$$

by induction of the size $\#(E)$ of S (in fact, the induction hypothesis states that the head operator of the term t corresponds to the connective, if it is unique, for which E is connected).

If $\#(E)<2$, then this is trivial: t is either $\mathbb{1}$ or an atom (given by the labelling function). Otherwise, by the previous lemma, there exists a connective U for which E is connected and not $\ddagger(U)$-connected. Let

$$\{C_1, \ldots, C_m\} = E/\sim_{\ddagger(U)}.$$

Then $1<m\leq\#(E)$, and for all i $(1\leq i\leq m)$ $S\lceil C_i\in\mathscr{X}(A)$ since the X-property is hereditary. Then, from the previous lemma, each C_i is connected for some connective V distinct from U. By induction hypothesis, there are terms $t_1, \ldots, t_m$ of $\mathscr{W}(A)$ such that

$$\forall i(1\leq i\leq m) \quad \mathscr{I}(t_i)\rightleftharpoons S\lceil C_i.$$

From the definition of the C_i's it cannot be the case that $e\ \ddagger(U)\ e'$ for some $e\in C_i$ and $e'\in C_j$ $(i\neq j)$. Suppose now that U is $\leq$ (the other cases where U is $\#$ or $\smile$ are similar and even simpler). Let us see that if $e<e'$ for some $e\in C_i$ and $e'\in C_j$, then, for all $e''\in C_j$, $e<e''$ whence $C_i\times C_j\subseteq<$: assume that $\exists e''\in C_j\ e''<e$; there exists a sequence of events of C_j which V-connects e' and e'', where V is either $\#$ or $\smile$. An easy induction on the length of such a sequence shows that we have the following consequence:

$$\exists e_0, e_1\in C_j \quad e_0<e<e_1 \text{ and } e_0\ V e_1.$$

But $e_0<e_1$ and $e_0\ V e_1$ is impossible. Therefore, we may assume that $\{C_1, \ldots, C_m\}$ is enumerated in such a way that $C_i\times C_{i+1}\subseteq<$. Then

$$S\rightleftharpoons(C_1;(\ldots;C_m)\ldots)$$
$$=(\mathscr{I}(t_1);(\ldots;\mathscr{I}(t_m))\ldots)=\mathscr{I}((t_1;(\ldots;t_m)\ldots)).$$

To conclude the proof of the theorem, we must show

$$t, t'\in\mathscr{N}(A) \Rightarrow \mathscr{I}(t)\rightleftharpoons\mathscr{I}(t')\Leftrightarrow t=_Y t'.$$

The proof of this last point is omitted. □

The characterization theorem gives also some indications on the nature of (infinite) A-LES's denoted by terms of $\mathbf{T}^{\mathrm{rec}}(A)$. It is easily seen that the poset $(\mathscr{X}(A)^\infty, \subseteq)$ is a coherent algebraic poset whose set of finite points is $(\mathscr{X}(A), \subseteq)$. Moreover, the operations of sequential composition, sum, and parallel composition (which are continuous) preserve these posets. Then it should be clear that the denotation $\mathscr{I}^\infty(p)$ of any term $p\in\mathbf{T}^{\mathrm{rec}}(A)$ is in $\mathscr{X}(A)^\infty$, that is,

$$\mathscr{T}^\infty(A)\subseteq\mathscr{X}(A)^\infty/\rightleftharpoons.$$

Recall that $\mathscr{T}^\infty(A)=\mathscr{I}(\mathbf{T}^{\mathrm{rec}}(A))$.

PART II: THE EXECUTION MODEL

4. Operational semantics

4.1. Transitions on labelled event structures

The interpretation equality $=_{\mathscr{I}}$ is too discriminating; from a behavioural point of view, we would like to identify for instance the terms $(a+b);c$ and $a;c+b;c$ as well as $p+p$ and p. Quite clearly, our system model for sequentiality, nondeterminism and concurrency—that is A-LES's—does not cope with the dynamic aspect of these programming concepts; we now want to devise an *execution model* for these constructs, that is, a notion of *computation*. Winskel has introduced (see [69]) such a notion for event structures, which he calls *configurations*. Configurations are "determinate prefixes" of ES's, that is, downwards closed and conflict-free subsets of events; this formalizes the ideas that an event cannot occur during a computation unless its causes have occurred, and that choices (i.e., conflicts) are resolved while a program computes. The execution model we look for is based upon a notion of computation which would be Winskel's notion of finite configuration if we had assumed the axiom of conflict heredity. Till now concurrency and conflict played similar rôles; the notion of computation will introduce an asymmetry, reflecting part of the "dynamic" nature of the sum. Our computations bear some analogy with processes of Petri nets [23, 26] or, more accurately, with Reisig's abstract processes [59].

Definition 4.1. Given an A-labelled event structure $S=(E, \leqslant, \#, \lambda)$ a *computation* of S is a structure $S\lceil F$ where

(i) F is a finite subset of E,

(ii) $S\lceil F$ is conflict-free: $e\in F \;\&\; e'\in F \Rightarrow \neg(e\# e')$,

(iii) $S\lceil F$ is closed under nonconflicting causes:

$$e\in F \;\&\; e'\leqslant e \;\&\; e'\notin F \;\Rightarrow\; \exists e''\in F\, e'\# e''\leqslant e.$$

We only allow finite computations, thus we cannot deal with *fairness*; an idea could be that fair computations are the—possibly infinite, but satisfying the axiom of finite causes—*maximal* computations, w.r.t. the ordering $\subseteq$.

Let us see some examples of computations: making an identification between terms and the structures they denote, $(a;c)$ and $(b;c)$ are computations of $((a+b);c)$. This example shows why we cannot assume that a computation is downwards closed (that is, $e\in F \;\&\; e'\leqslant e \Rightarrow e'\in F$) since otherwise no computation of $(a+b);c$ could contain c. In the structure $((a+b);c)$, a and b are causes of c, but c cannot occur unless a choice has been made between a and b. The computations of the structure denoted by $\mu x.(a\|x)$ are a, $(a\|a), \ldots, (a\|\cdots\|a)$, and so on.

Since computations are deterministic, they are just A-labelled posets. In this paper we restrict our attention to A-LES's of $\mathscr{X}(A)^{\infty}$. The computations of such structures

satisfy the X-property. Since the ∇-freeness property is vacuously true for conflict-free structures, computations are in fact, by Lemma 3.2, N-free A-labelled posets. Let us denote by $\mathcal{P}(A)$ the set of these computations; we shall give the name of *action* to an isomorphism class of computations, element of $\mathcal{D}(A) = \mathcal{P}(A)/\rightleftharpoons$. Then $\mathcal{D}(A)$ is exactly the set of what Pratt and Gischer [24] call finite *N-free pomsets.* From a theorem of Grabowski and Gischer, $\mathcal{D}(A)$ is the free "*dioid*" on A (Grabowski calls it "double monoid"), which is the same as a trioid but without sum. All that means that actions are denoted by terms built without sum, up to the equational theory Δ whose axioms are A0, A1, U0, U1 and C1. We denote by $\mathbf{D}(A)$ the set of these deterministic terms, which we shall abusively call *actions* (cf. Appendices A and B for the syntax and the equational theory). As a matter of fact, we extend Milner's idea [46, 48] that actions should be elements of a commutative monoid, or more generally elements of a synchronization algebra, as proposed by Winskel [69, 70].

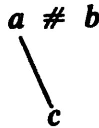
From a computation $P = S\lceil F$ of S we build a structure called the *residual of S by P* which is

$$(S/P) \stackrel{\text{def}}{=} S\lceil(E-(F\cup\#(F)))$$

where

$$\#(F)=\{e \mid \exists e'\in F\, e'\#e\}.$$

The structure (S/P) is "what remains of S after performing P while resolving the conflicts". This definition only makes sense for structures S which satisfy the triangle-freeness property (or the axiom of conflict heredity); for instance, b is a computation of the event structure ∇′



(where $c \smile b$) and its residual is c, but this is clearly absurd. Anyway, the definition works well for the event structures we are interested in, that is, structures satisfying the X-property. Clearly, $S \in \mathcal{X}(A)^\infty$ implies $(S/P) \in \mathcal{X}(A)^\infty$.

We are now ready to introduce the main definition which brings a structure of transition system on A-LES's. Let us recall the terminology: a (*labelled*) *transition system* $\Sigma = (Q, C, \theta)$ is a structure where

(i) Q is the set of *states*,
(ii) C is the set of *computations*,
(iii) $\theta \subseteq Q \times C \times Q$ is the *transition relation.* As notation, $p \xrightarrow[\theta]{\gamma} p'$ or $p \rightarrow^{\gamma}_{\theta} p'$ will denote $(p, \gamma, p') \in \theta$.

Definition 4.2. The transition relation η between A-labelled event structures is given by

$$S \xrightarrow[\eta]{P} S' \stackrel{\text{def}}{\Leftrightarrow} P \text{ is a computation of } S \text{ and } S' = (S/P).$$

Here one can see some analogy with the construction h_1 before h_2 gives h of Degano and Montanari [17] if one reads it as $h \rightarrow^{h_1} h_2$. Similar definitions havealso been introduced by Grabowski [27] and Reisig [59] who define transitions labelled by pomsets on the state space (markings) of a Petri net.

For instance, still using terms in place of structures, we have

$$(a;b) \xrightarrow[\eta]{a} b, \qquad (a\|b) \xrightarrow[\eta]{a} b, \qquad (a\|b) \xrightarrow[\eta]{(a\|b)} \mathbb{1}$$

$$(a;(b;c+d)\|a) \xrightarrow[\eta]{(a;b\|a)} c.$$

One always has $S \rightarrow^{\mathbb{1}}_{\eta} S$, and we shall interpret the action $\mathbb{1}$ as *identity*. On the other hand, as a residual of a computation, $\mathbb{1}$ means *termination*, and we shall read $S \rightarrow^{P}_{\eta} \mathbb{1}$ as "S terminates in performing the computation P". Note that an infinite structure may very well terminate in performing a (finite) computation; for instance, we have

$$\mu x.(a;(x+b)) \xrightarrow[\eta]{a;\cdots;a;b} \mathbb{1}$$

(the term $\mu x.(a;(x+b))$ has an interpretation similar to that of $\mu x.(a;(b\|x))$—cf. Section 2.3—with conflict in place of concurrency).

One may remark that from the definition, a computation of an A-LES S cannot introduce causal dependencies which would not be already present in S. For instance, $(a;b)$ is not a computation of $(a\|b)$. On the other hand, we have the following lemma.

Lemma 4.3. *If $P;Q$ is a computation of S, then $P' = P;\mathbb{1}$ is a computation of S, $Q' = \mathbb{1};Q$ is a computation of (S/P') and $(S/(P;Q)) = ((S/P')/Q')$.*

Proof. If $P;Q$ is a computation of S, then there exists a subset F of the set E of events of S such that $P;Q = (S\lceil F)$. Moreover, $F = F_0 \dot{\cup} F_1$ with $P' = S\lceil\{0\}F_0$ and $Q' = S\lceil\{1\}F_1$. We let $F_0' = \{0\}F_0$ and $F_1' = \{1\}F_1$. It is clear that P' is a computation of S, and that $Q' \subseteq (S/P')$. Let us show that $(S/P')\lceil F_1'$ is closed under nonconflicting causes: if $e \in F_1'$, $e' \in E - (F_0' \cup \#(F_0') \cup F_1')$, and $e' \leqslant e$, then $\exists e'' \in F_0' \cup F_1'\ e' \# e'' \leqslant e$ since $P;Q$ is a computation of S; but it cannot be the case that $e'' \in F_0'$ since otherwise we would have $e' \in \#(F_0')$; hence $e' \in F_1'$. Then Q' is a computation of (S/P'), and since $\#(F_0' \cup F_1') = \#(F_0') \cup \#(F_1')$ we have $(S/(P;Q)) = ((S/P')/Q')$. □

We could then say that in our behavioural semantics causality implies temporal ordering for we have, by the previous lemma,

$$S \xrightarrow[\eta]{P;Q} S' \Rightarrow \exists S''\, \exists P' \rightleftharpoons P\, \exists Q' \rightleftharpoons Q\ \ S \xrightarrow[\eta]{P'} S'' \xrightarrow[\eta]{Q'} S'.$$

But the converse is false: although $(a\|b) \rightarrow^{a}_{\eta} b \rightarrow^{b}_{\eta} \mathbb{1}$, we do not have $(a\|b) \rightarrow^{a;b}_{\eta} \mathbb{1}$. Thus our semantics makes a strong distinction between *sequence of transitions* and

"*transitions of a sequence*"—compare with the CCS "action" $a.p$. This distinction does not hold in the model of Grabowski where a sequence of computations of a marked net is still a computation of this net. Another point is that our execution model is free from any assumption of *global time*: even if we think about a transition step as occurring in a time unit, this is not related to any hypothesis about the *duration* of the atoms. For instance, $(a;b\|c)$ is a possible computation; in some sense we could say, as in the programming language ESTEREL [4], that "the sequential composition ; takes no time".

One may also note that the behavioural interpretation of parallel composition is not interleaving, but generalizes it. Our semantics of parallel composition is also a generalization of the MEIJE "asynchronous" operator [2, 5] introduced by Milner in [45] (see also [48]). This asynchronous concurrency is related to the notion of "firing step" of Petri nets [68, 59], where one fires a multiset of concurrent transitions—this is shown in [6].

4.2. Transitions on terms

Since we are interested in labelled event structures denoted by terms of $\mathbf{T}^{\text{rec}}(A)$, an obvious question is: is there any syntactic notion of transition which reflects the semantic one? In fact the (positive) answer is rather simple. We shall see that the intended *operational semantics* for terms is given by the transition relation ρ, defined as the least subset of $\mathbf{T}^{\text{rec}}(A)\times\mathbf{D}(A)\times\mathbf{T}^{\text{rec}}(A)$ satisfying the following clauses or *rules*:

E0: *identity*

$$\vdash p \xrightarrow{\mathbb{1}} p,$$

E1: *atom*

$$a\in A \vdash a \xrightarrow{a} \mathbb{1},$$

E2.1: *sequential composition* 1

$$p \xrightarrow{u} p' \vdash (p;q) \xrightarrow{u} (p';q),$$

E2.2: *sequential composition* 2

$$p \xrightarrow{u} p' =_{\Theta} \mathbb{1},\ q \xrightarrow{v} q' \vdash (p;q) \xrightarrow{(u;v)} q',$$

E3: *parallel composition*

$$p \xrightarrow{u} p',\ q \xrightarrow{v} q' \vdash (p\|q) \xrightarrow{(u\|v)} (p'\|q'),$$

E4.1: *sum* 1

$$p \xrightarrow{u} p' \;\&\; u \neq_{\Theta} \mathbb{1} \vdash (p+q) \xrightarrow{u} p',$$

E4.2: *sum* 2

$$q \xrightarrow{v} q' \ \&\ v \neq_{\Theta} \mathbb{1} \vdash (p+q) \xrightarrow{v} q',$$

E5: *fixpoint*

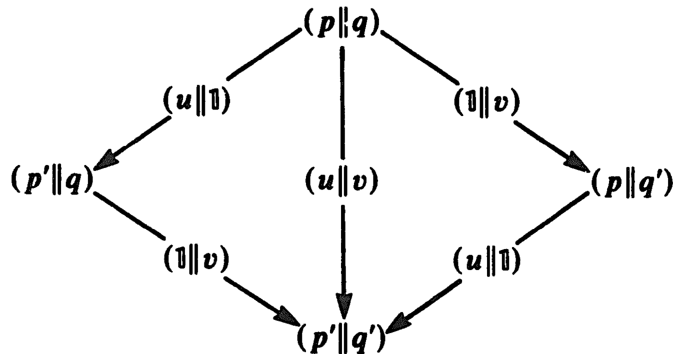
$$p[\mu x.p/x] \xrightarrow{u} p' \vdash \mu x.p \xrightarrow{u} p'.$$

(E stands for "execution"). Note that $r =_{\Theta} \mathbb{1}$ can be proved or disproved using only the axioms U0 to U2. Then this is nearly syntactical equality—which does not mean testing deadlock or termination of an algorithm! Moreover, the rules only use the termination test $r =_{\Theta} \mathbb{1}$ for finite closed terms, that is, $r \in \mathbf{T}(A)$ since the formation of $(p;q)$ requires $p \in \mathbf{T}(A)$. One may observe that no rule can introduce a sum or a recursive definition in the actions, while sequential and parallel composition are introduced by E2.2 and E3; then ρ-transitions are labelled by terms denoting finite labelled posets (a similar idea was used for CCS by Degano, De Nicola and Montanari in [19]).

Since ρ is the *least* relation satisfying the given clauses, a transition $p \rightarrow_{\rho}^{u} p'$ cannot hold unless it has a proof of construction according to these rules. For instance, we have

$$\text{E2.2:}\ \cfrac{\text{E4.1:}\ \cfrac{\text{E1:}\ \cfrac{}{a \xrightarrow[\rho]{a} \mathbb{1}}}{(a+b) \xrightarrow[\rho]{a} \mathbb{1}} \qquad \text{E1:}\ \cfrac{}{c \xrightarrow[\rho]{c} \mathbb{1}}}{((a+b);c) \xrightarrow[\rho]{(a;c)} \mathbb{1}}$$

It should be clear from the rules that parallel composition is an *asynchronous* operator, as in MEIJE since we always have, if $p \rightarrow_{\rho}^{u} p'$ and $q \rightarrow_{\rho}^{v} q'$:



Now we state the *adequacy* result establishing the correspondence between transitions on terms and transitions on event structures.

Theorem 4.4. (Adequacy). *For all* $r \in \mathbf{T}^{\text{rec}}(A)$

(i) $r \underset{\rho}{\overset{w}{\to}} s \;\Rightarrow\; \exists W \mathcal{I}(w) \rightleftharpoons W \; \mathcal{I}^{\infty}(r) \underset{\eta}{\overset{W}{\to}} (\mathcal{I}^{\infty}(r)/W) \;\&\; \mathcal{I}^{\infty}(s) \rightleftharpoons (\mathcal{I}^{\infty}(r)/W)$,

(ii) $\mathcal{I}^{\infty}(r) \underset{\eta}{\overset{W}{\to}} S \;\Rightarrow\; \exists w \mathcal{I}(w) \rightleftharpoons W \; \exists s \mathcal{I}^{\infty}(s) \rightleftharpoons S \; r \underset{\rho}{\overset{w}{\to}} s.$

The first point states the *validity* of the rules E0 to E5 for the semantical notion of transition, the second one states their *completeness.*

Proof. Let us show the first point: the argument we use is an induction on the proof of the transition $r \to_{\rho}^{w} s$. The point is trivial if this transition is an instance of E0 or E1. If the last rule of its proof is E2.1, then $r = (p;q)$, $s = (p';q)$. By induction hypothesis, there exists a W such that $\mathcal{I}(w) \rightleftharpoons W$ and $\mathcal{I}^{\infty}(p) \to_{\eta}^{W} P'$ where P' is such that $\mathcal{I}^{\infty}(p') \rightleftharpoons P'$. Then it is clear that $W' = W;\mathbb{1}$ is a computation of $\mathcal{I}^{\infty}(p);\mathcal{I}^{\infty}(q)$, and that

$$(\mathcal{I}^{\infty}(p);\mathcal{I}^{\infty}(q))/W' = (\mathcal{I}^{\infty}(p)/W);\mathcal{I}^{\infty}(q) \rightleftharpoons \mathcal{I}^{\infty}(p');\mathcal{I}^{\infty}(q)$$

(we leave it to the reader to check this).

Assume now that $r \to_{\rho}^{w} s$ is proved by means of E2.2. Then $r = (p;q)$, $s = q'$, $w = (u;v)$, and there exist $U \rightleftharpoons \mathcal{I}(u)$, $V \rightleftharpoons \mathcal{I}(v)$, and $Q' \rightleftharpoons \mathcal{I}^{\infty}(q')$ such that $P \to_{\eta}^{U} \mathbb{1}$ and $Q \to_{\eta}^{V} Q'$ (with $P = \mathcal{I}^{\infty}(p)$ and $Q = \mathcal{I}^{\infty}(q)$). Let us show that $U;V$ is a computation of $R = P;Q$: the only point to verify is that $U;V$ is closed under nonconflicting causes. So let e and e' be events of $U;V$ and R respectively such that $e' \leq e$ and e' is not an event of $U;V$. Let us assume that e is an event of $V' = \mathbb{1};V$ and that e' is an event of $P;\mathbb{1}$ (the other possible cases being trivial); then, since $P/U = \mathbb{1}$, there must exist an event e'' of $U' = U;\mathbb{1}$ such that $e' \# e''$, and obviously $e'' \leq e$, so we are done. Then, by Lemma 4.3,

$$(R/(U;V)) = ((R/U')/V') = ((P;Q)/U')/V' = ((P/U);Q)/V' \rightleftharpoons Q/V = Q'.$$

We omit the proof of the validity of the rules E3, E4.1 and E4.2. One should note that in the last two cases the semantical computation W is not equal to $\mathcal{I}(w)$—but isomorphic to it—, as in the case of E2.1, and that the residual of this computation is not equal to $\mathcal{I}^{\infty}(s)$—but isomorphic to it—, as in the case of E2.2. When $r \to_{\rho}^{w} s$ is proved by means of E5, one directly uses the induction hypothesis since $\mathcal{I}^{\infty}(p[\mu x.p]) = \mathcal{I}^{\infty}(\mu x.p)$ by definition. This ends the proof of the first point of the theorem.

In order to establish the completeness part, one must observe that a computation W of $R = \mathcal{I}^{\infty}(r)$ is a computation of $\mathcal{I}(\varpi(\kappa^{n}(r)))$ for some n—because any computation is finite. We thus associate with a transition $\mathcal{I}^{\infty}(r) \to_{\eta}^{W} S$ the pair $(k, |r|)$ of integers, where

$$k = \begin{cases} 0 & \text{if } W = \mathbb{1}, \\ n+1 & \text{otherwise, where } n \text{ is the least integer such} \\ & \text{that } W \text{ is a computation of } \mathcal{I}(\varpi\kappa^{n}(r)) \end{cases}$$

and $|r|$ is the *size* of r, defined by

(i) $|\mathbb{1}| = 1$;

(ii) $|a| = 1$ for all $a \in A$;

(iii) $|(p;q)| = |p| + |q|$, $|(p\|q)| = |p| + |q|$ and $|(p+q)| = |p| + |q|$;

(iv) $|\mu x.p| = |p|$.

Then the proof proceeds by induction on the pairs $(k, |r|)$, ordered lexicographically: when $k = 0$, the proof is trivial (one just uses E0). Otherwise $(k > 0)$, one uses an induction on the structure of the term r. When $r = \mu x.p$, since $\mathcal{S}^\infty(p[\mu x.p]) = \mathcal{S}^\infty(r)$, one has $\mathcal{S}^\infty(p[\mu x.p]) \to_\eta^W S$, and the pair associated with this transition is $(k-1, |p[\mu x.p]|)$ (since $W \neq \mathbb{1}$). One therefore applies directly the induction hypothesis. We omit the straightforward proof of the other cases (note that when $r = (p;q)$ and $W \rightleftharpoons U;V$ for some computations U and V of $\mathcal{S}^\infty(p)$ and $\mathcal{S}^\infty(q)$, then $p \to_\rho^u p'$ where $\mathcal{S}^\infty(p') = \mathbb{1}$; then $p' =_\Theta \mathbb{1}$ since $p \in \mathbf{T}(A)$, so that we can use E2.2). □

We could have split rule E3 of parallel composition into the following three:

E3.1: $p \xrightarrow{u} p' \vdash (p\|q) \xrightarrow{u} (p'\|q)$,

E3.2: $p \xrightarrow{u} p',\ q \xrightarrow{v} q' \vdash (p\|q) \xrightarrow{(u\|v)} (p'\|q')$,

E3.3: $q \xrightarrow{v} q' \vdash (p\|q) \xrightarrow{v} (p\|q')$.

It is easily shown that the resulting transition system is semantically equivalent to ρ. The rules E3.1 and E3.3 are the rules of *interleaving*, while E3.2 is the rule of Milner's *synchronous product*. Using these rules in place of E3 we can make a classification:

(1) by taking all the rules except E0, E2.2 and E3.2, we get a CCS-like transition system [44, 46]: since we have precluded the rules introducing the empty computation, sequential composition and parallel composition of computations, the actions are simply atoms;

(2) if we use all the rules except E0 and E2.2, we get a MEIJE-like transition system [2, 5], where the actions are nonempty multisets of atoms—what one builds from atoms using the associative and commutative parallel composition;

(3) if we use all the rules except E0, E2.2, E3.1 and E3.3, we get an SCCS-like transition system [46], with synchronous parallel composition.

It should be clear that these transition relations correspond to semantical notions of transition, that is, transitions on event structures. In the first case, a computation is $S\lceil\{e\}$ where e is a minimal event of S; in the second one, a computation is $S\lceil F$ where F is a nonempty finite set of concurrent minimal events, while in the last one, F must be a maximal such set. We could also consider the transition we get by precluding E3.2 (together with E0 or not) but allowing sequential composition in computations (that is, E2.2); although this corresponds to a notion of computation of event structures ($S\lceil F$ with F totally ordered), this does not seem to fit in with any known semantics.

5. Semantics

5.1. Equipollence

Relative to any transition system one may define various kinds of semantics and equivalences, among which the better known are (see [56] for a survey, and [8, 20] for a comparison):

(1) *trace* semantics such as used by Hoare [32];
(2) *failure* semantics of [9] (other equivalent definitions are possible);
(3) *testing* equivalence of Hennessy and De Nicola [31];
(4) *logical equivalences*, induced for instance by *trace logics* (cf. [30, 8]);
(5) Park and Milner's notion of *bisimulation* [51, 46].

For example, Taubner and Vogler [64] have studied the failure semantics of step transition systems; De Nicola et al. [1] have adapted testing equivalence to our notion of transition on LES's.

Here we adopt the notion of bisimulation, with a slight variation however. A bisimulation is a relation over states of a transition system $\Sigma = (Q, C, \theta)$ such that two related states have similar behaviours. But we also need a relation on actions: some actions ought to be regarded as the same. For instance the reader may have remarked that strictly speaking, $(a\|b) \rightarrow^{a} b$ is not a valid transition—neither for η nor for ρ! We might have written it $(a\|b) \rightarrow^{(a\|\mathbb{1})} (\mathbb{1}\|b)$. We have seen similar technicalities in the adequacy theorem. Therefore, we would like to consider transitions labelled by actions, that is, labelled by isomorphism classes of computations—or equivalently by elements of $\mathbb{D}(A)/=_{\Delta}$. We shall use the following terminology (see [8, 5]): let $R \subseteq Q \times Q$ be a relation over states and $H \subseteq C \times C$ be a relation over computations. The pair (R, H) is

(i) *invariant* with respect to θ if and only if it satisfies

$$p\,R\,q \text{ and } p \xrightarrow[\theta]{\gamma} p' \Rightarrow \exists \gamma'\ \gamma\,H\,\gamma'\ \exists q'\ p'\,R\,q' \text{ and } q \xrightarrow[\theta]{\gamma'} q';$$

(ii) a *bisimulation* (w.r.t. θ) if it is an invariant pair of symmetric relations;
(iii) an *equisimulation* if it is an invariant pair of equivalence relations.

The invariance property is usually drawn

$$\begin{array}{ccc} p & \text{——} R \text{——} & q \\ | & & \vdots \\ \gamma & \cdots H \cdots & \gamma' \\ \downarrow & & \vdots \\ p' & \cdots R \cdots & q' \end{array}$$

The following fact is standard.

Lemma 5.1. *Given a transition system $\Sigma = (Q, C, \theta)$ and G an equivalence relation*

over C, let us define

$$p \asymp_\theta^G q \overset{\text{def}}{\Leftrightarrow} \exists(R, H) \text{ bisimulation such that } p\,R\,q \;\&\; H \subseteq G.$$

Then $(\asymp_\theta^G, G)$ *is an equisimulation and it is the coarsest among the equisimulations* (R, H) *such that* $H \subseteq G$.

Proof. The only point to check is that the composition $(R \circ R', H \circ H')$ of invariant pairs is itself invariant—just draw it. □

We shall call this relation $\asymp_\theta^G$ the *equipollence* with respect to θ and G.

Let us return to the two systems we introduced in the previous section, $(\mathbf{T}^{\text{rec}}(A), \mathbf{D}(A), \rho)$ and $(\mathscr{X}(A)^\infty, \mathscr{P}(A), \eta)$. As we said, we are in fact interested in transitions labelled by *actions*. Therefore, the computations of these systems will be considered up to the equivalence relations $=_\Delta$ (or equivalently $=_\wp$) and $\rightleftharpoons$ respectively. The equipollence $\asymp_\rho^{=_\Delta}$ is what we regard as defining the *semantic equality* of terms. Thus we just use $\asymp_\rho$ or $\asymp$ to denote it. For instance, the three terms $(a\|b)$, $(a;b)+(b;a)$ and $(a;b)+(a\|b)+(b;a)$ are pairwise distinct with respect to $\asymp$ since the first cannot perform the action $(a;b)$ whereas the second cannot perform $(a\|b)$. Another example is

$$(a;b\|c) \not\asymp (a\|c);b + a;(b\|c).$$

Let us return for a while to the classification of transition systems we made in the previous section. Using the subscripts CCS, SCCS and MEIJE just to suggest the analogy with the corresponding calculi, we shall denote:

(1) $\asymp_{\text{CCS}}$ the equipollence relative to the least transition relation satisfying E1, E2.1, E3.1, E3.3 and E4.1 to E5 (with a trivial equality for computations);

(2) $\asymp_{\text{MEIJE}}$ the equipollence relative to the least transition relation satisfying E1, E2.1, E3.1, E3.2, E3.3 and E4.1 to E5 (with $=_\Delta$ as the equality for computations);

(3) $\asymp_{\text{SEQ}}$ the equipollence relative to the least transition relation satisfying E1, E2.1, E2.2, E3.1, E3.3 and E4.1 to E5 (idem);

(4) $\asymp_{\text{SCCS}}$ the equipollence relative to the least transition relation satisfying E1, E2.1, E3.2 and E4.1 to E5 (idem).

Then we have, for all p, q in $\mathbf{T}^{\text{rec}}(A)$,

$$p \asymp q \Rightarrow p \asymp_{\text{MEIJE}} q \Rightarrow p \asymp_{\text{CCS}} q,$$

$$p \asymp q \Rightarrow p \asymp_{\text{SEQ}} q \Rightarrow p \asymp_{\text{CCS}} q.$$

The converse implications are false, and $\asymp_{\text{MEIJE}}$ and $\asymp_{\text{SEQ}}$ are incomparable:

(1) $(a\|b) \asymp_{\text{CCS}} a;b+b;a$, but this is false in the other equipollences;

(2) $(a\|b) \asymp_{\text{MEIJE}} a;b+(a\|b)+b;a$, but this is false for $\asymp_{\text{SEQ}}$, $\asymp_{\text{SCCS}}$ and $\asymp$;

(3) $a;b+b;a \asymp_{\text{SEQ}} a;b+(a\|b)+b;a$, but this is false for $\asymp_{\text{MEIJE}}$, $\asymp_{\text{SCCS}}$ and $\asymp$.

The situation of $\asymp_{\text{SCCS}}$ is somewhat special: for instance,

$$(a+b)\|c \asymp_{\text{SCCS}} (a\|c)+(b\|c) \quad \text{and} \quad (a;b\|c) \asymp_{\text{SCCS}} (a\|c);b,$$

but these are false in the other equipollences. What is common to CCS, MEIJE and SCCS is that they all assume a global time according to which the sequencing ; corresponds to a "clock interrupt".

Since the equipollence $\asymp$ is a strong bisimulation in Milner's sense (cf. [44, 46]), it has some nice properties with respect to the algebraic structure in the following proposition.

Proposition 5.2. *The equipollence* $\asymp$ *is a congruence of the trioid* $\mathbf{T}^{\mathrm{rec}}(A)$:

$$p \asymp p' \text{ and } q \asymp q' \Rightarrow \begin{cases} (p;q) \asymp (p';q'), \\ (p+q) \asymp (p'+q'), \\ (p \| q) \asymp (p' \| q'). \end{cases}$$

The (easy) proof is omitted. Now we want to relate the syntactic equipollence $\asymp_\rho$, and the semantic one $\asymp_\eta^{\rightleftharpoons}$ that we denote simply by $\asymp_\eta$. To this end we introduce a notion of *morphism* of transition systems.

Definition 5.3. Let $\Sigma = (Q, C, \theta)$ and $\Sigma' = (Q', C', \theta')$ be two transition systems. A morphism from Σ to Σ' is a pair (φ, ψ) of mappings $\varphi : Q \to Q'$ and $\psi : C \to C'$ which satisfies:

(i) soundness:

$$q \xrightarrow[\theta]{\gamma} s \Rightarrow \varphi(q) \xrightarrow[\theta']{\psi(\gamma)} \varphi(s);$$

(ii) completeness:

$$\varphi(q) \xrightarrow[\theta']{\psi(\gamma)} s \Rightarrow \exists \gamma' \exists r \, \psi(\gamma') = \psi(\gamma) \;\&\; \varphi(r) = s \;\&\; q \xrightarrow[\theta]{\gamma'} r.$$

Various kinds of morphisms of transition systems may be found in the literature (see [5] for some references). They are mainly introduced to formalize the notion of "reduction" by means of which one can verify properties of systems. Here, as in [5, 14], we want morphisms to be strongly related to equisimulations. Every equisimulation determines a quotient transition system, and the projection onto the quotient is a morphism. For instance, since $(\rightleftharpoons, \rightleftharpoons)$ is clearly an equisimulation of the system $(\mathscr{X}(A)^\infty, \mathscr{P}(A), \eta)$, we can define a quotient transition relation $\hat{\eta}$ and we have (still using terms in place of the structures)

$$[\![(a \| b)]\!] \xrightarrow[\hat{\eta}]{[\![a]\!]} [\![b]\!].$$

As a matter of fact we have already met a morphism of transition systems since the adequacy theorem actually states that the pair of mappings

$$\mathscr{S} : \mathbf{T}^{\mathrm{rec}}(A) \to \mathscr{X}(A)^\infty/\rightleftharpoons \quad \text{and} \quad \mathscr{S} : \mathbf{D}(A) \to \mathscr{D}(A)$$

is a morphism from $(\mathbf{T}(A), \mathbf{D}(A), \rho)$ to $(\mathscr{X}(A)^\infty/\rightleftharpoons, \mathscr{D}(A), \hat{\eta})$.

We shall not develop at length the theory of morphisms. We just mention that it would allow us to prove for instance that $(=_\Theta, =_\Delta)$ is an equisimulation of $(\mathbf{T}(A), \mathbf{D}(A), \rho)$, whence

$$p =_\Theta q \Rightarrow p \asymp_\rho q.$$

In fact this last implication is true for terms of $\mathbf{T}^{\text{rec}}(A)$. Moreover, we could prove that there is an exact correspondence between the syntactic and sematic equipollences as follows.

Proposition 5.4

$$p \asymp_\rho q \Leftrightarrow \mathcal{I}(p) \asymp_{\hat\eta} \mathcal{I}(q) \Leftrightarrow \mathcal{I}(p) \asymp_\eta \mathcal{I}(q).$$

The main interest we have in these facts is that they justify (a posteriori!) the use we have made of an overloaded execution arrow: we can now write $p \to^u q$, regardless of the fact that p, u and q are terms, event structures or equivalence classes for some equisimulation.

5.2. Axiomatization

In this section we plan to set up a "proof theory" of $\asymp$—*for finite terms.* It should be clear that η-equipollence of elements of $\mathcal{P}(A)$ is exactly $=$, for

$$P \in \mathcal{P}(A) \Rightarrow (P \xrightarrow[\eta]{Q} \mathbb{1} \Leftrightarrow Q = P).$$

Thus any intended axiomatization essentially states properties of the sum. As a matter of fact there is a standard way to solve the problem, by means of *sumforms* as Hennessy and Milner have shown in [30] which we will briefly recall now. For any set C of actions let $\mathbf{K}(C)$ be the set of terms built according to the following rules.

Sumforms. (i) $\mathbb{1}$ is a term;

(ii) for every $\gamma \in C$ if p is a term, then $\gamma \bullet p$ is a term;

(iii) if p and q are terms, then so is $(p+q)$.

Let Ψ be the theory whose axioms are A2, U2, C2 (cf. Appendix B), and

I: $\quad (p+p)=p$

and μ the least transition relation on $\mathbf{K}(C)$ given by the rules

E0′: $\quad \vdash \gamma \bullet p \xrightarrow{\gamma} p,$

E4.1′: $\quad p \xrightarrow{\gamma} p' \vdash (p+q) \xrightarrow{\gamma} p',$

E4.2′: $\quad q \xrightarrow{\gamma} q' \vdash (p+1) \xrightarrow{\gamma} q'.$

Then the Hennessy–Milner theorem roughly states the following.

Theorem 5.5. *Any state of a finite acyclic transition system on C is denoted by a term of $\mathbb{K}(C)$. For such terms*

$$p \asymp_{\mu} q \Leftrightarrow p =_{\Psi} q$$

(*where $\asymp_{\mu}$ is the equipollence relative to equality of computations*).

From this result, we just have to find a suitable translation from $\mathbb{T}(A)$ to $\mathbb{K}(C)$ (that is, an expansion of finite terms into finite acyclic transition systems) in order to solve our axiomatization problem. A first step is to extend our set of terms to $\mathbb{T}'(A)$ which is built as $\mathbb{T}(A)$ but with the additional formation rule:

(iii) if $\gamma \in \mathbb{D}'(A)$ and $p \in \mathbb{T}'(A)$, then $(\gamma \bullet p) \in \mathbb{T}'(A)$,

where $\mathbb{D}'(A)$ is the set of terms built from A using ; and $\|$ (without $\mathbb{1}$). We also extend the transition relation ρ to ρ' with the supplementary rule E0′. Axiom A2 allows us to use an ambiguous notation $\sum_i p_i$ for a (finite) sum of terms. Then our axiomatization is as follows: let Φ be the (heterogeneous) theory whose axioms are those of Θ (cf. Appendix B) plus I and (omitting some parentheses)

B1: $\quad a \bullet \mathbb{1} = a \quad$ for $a \in A$,

B2: $\quad \left(\sum_i \gamma_i \bullet p_i\right);q = \sum_i ((\gamma_i \bullet p_i);q)$,

B3: $\quad (\gamma \bullet \mathbb{1});\left(\sum_j \beta_j \bullet q_j\right) = \gamma \bullet \left(\sum_j \beta_j \bullet q_j\right) + \sum_j (\gamma;\beta_j) \bullet q_j$,

B4: $\quad \left(\gamma \bullet \left(\sum_i \gamma_i \bullet p_i\right)\right);q = \gamma \bullet \left(\sum_i (\gamma_i \bullet p_i);q\right)$,

B5: $\quad \left(\sum_i \gamma_i \bullet p_i \| \sum_j \beta_j \bullet q_j\right) = \sum_i \gamma_i \bullet \left(p_i \| \sum_j \beta_j \bullet q_j\right) + \sum_{i,j} ((\gamma_i \| \beta_j) \bullet (p_i \| q_j))$

$$+ \sum_j \beta_j \bullet \left(\sum_i \gamma_i \bullet p_i \| q_j\right).$$

Theorem 5.6 (Axiomatization). *Let $=_{\Phi}$ be the congruence of algebra generated by Φ. Then the pair $(=_{\Phi}, =_{\Delta})$ is invariant with respect to ρ'. Moreover, for each $p \in \mathbb{T}(A)$ there exists an $r \in \mathbb{K}(\mathbb{D}'(A))$ such that $p =_{\Phi} r$. Therefore, for p, $q \in \mathbb{T}(A)$, $p \asymp_{\rho} q \Leftrightarrow p =_{\Phi} q$.*

Proof (*outline*). The first statement, which implies *soundness*, namely $p =_{\Phi} q \Rightarrow p \asymp_{\rho'} q$, can be shown by a straightforward case inspection. More precisely, one shows that for each pair p, q of terms of $\mathbb{T}'(A)$ such that $p = q$ or $q = p$ is (an instance of) an axiom of Φ, and for each transition $p \to^{u}_{\rho'} p'$, there exist $v =_{\Delta} u$ and $q' =_{\Phi} p'$ such that $q \to^{v}_{\rho'} q'$; this is proved by induction on the proof of the transition $p \to^{u}_{\rho'} p'$.

Then one shows that the same fact holds for pairs p, q such that $p =_\Phi q$, by induction of the definition of $=_\Phi$—recall that $=_\Phi$ is the least relation on $\mathbf{T}'(A)$ containing the (instance of the) axioms of Φ and satisfying

$$p =_\Phi p' \text{ and } q =_\Phi q' \Rightarrow \begin{cases}(p;q) =_\Phi (p';q'),\\ (p+q) =_\Phi (p'+q'),\\ (p\|q) =_\Phi (p'\|q').\end{cases}$$

For the second point, we first extend the notion of size (cf. Subsection 4.2) to terms of $\mathbf{T}'(A)$ by $|\gamma \bullet p| = |\gamma| + |p|$. Then one can prove by induction on the size $|p|$ of $p \in \mathbf{T}'(A)$ that such a term is convertible by means of the axioms B into a *normal form*, which is here either $\mathbb{1}$ or a term $\sum_i \gamma_i \bullet p_i$ where each p_i is again a normal form. A consequence is *completeness*: $p \asymp_{\rho'} q \Rightarrow p =_\Phi q$. Let us show this point: we know that there exist r and $s \in \mathbf{K}(\mathbf{D}'(A))$ such that $p =_\Phi r$ and $q =_\Phi s$; then $p \asymp_{\rho'} r$ and $q \asymp_{\rho'} s$. Therefore, if $p \asymp_{\rho'} q$, we have $r \asymp_{\rho'} s$, whence $r =_\Phi s$ by the Hennessy-Milner theorem, and this implies $p =_\Phi q$ (note that Φ contains the equality theory Δ for the actions, which is needed to apply Hennessy-Milner's theorem). □

One could have the idea that this result expresses a reduction of concurrency to sequential nondeterminism; however, this is not quite right since actions are posets irreducibly involving parallelism. So the expansion theorem is not so bad. From a semantical point of view, the technique we used is still unsatisfactory since it gives no indication of how one could describe the equipollence classes of A-LES's. Nevertheless, our present purpose is achieved: we can prove semantic equalities of terms, such as the distributivity property

$$(a+b);c \asymp a;c+b;c.$$

A proof is

$$\begin{aligned}(a+b);c &= (a \bullet \mathbb{1} + b \bullet \mathbb{1});c && \text{(B1)}\\ &= (a \bullet \mathbb{1});c + (b \bullet \mathbb{1});c && \text{(B2)}\\ &= a;c+b;c && \text{(B1).}\end{aligned}$$

We can also prove

$$(a\|(b+c)) + (a\|b) + ((a+c)\|b) \asymp (a\|(b+c)) + ((a+c)\|b)$$

and other absorption phenomena (r is absorbed by p if $p+r \asymp p$, cf. [13, 15]). This example can be arbitrarily complicated (see [13, 15]), so that the existence of a finite axiomatization without extending the syntax or introducing an absorption preorder is doubtful. Note that it can be proved that our equipollence is weaker than the notion of distributed bisimulation of [13, 15]. Hennessy has found an example (of absorption) which proves that it is strictly weaker, namely

$$(a\|b+c) + a;(b+c) + (a\|b) + (a\|c) \asymp a;(b+c) + (a\|b) + (a\|c)$$

but these two equipollent terms are not d-bisimilar. This example shows that in our semantics, an agent—like a in $(a \| b+c)$—cannot in any way "know" that he is concurrent with a choice. This contrasts also with the generalized pomset bisimulation of Van Glabbeek and Vaandrager [25]. Further work should be undertaken to thoroughly compare various approaches to the theory of true concurrency, and especially [13], [19], [25], and [50].

PART III. THE OPERATION MODEL

6. Processes and operation

6.1. Operation of programs on objects

The aim of this section is to introduce a notion of operation for processes: we want to set up a formalism describing how processes use and change data. We begin with a brief account of the theory of programming languages semantics. In this theory (see [42] for instance), the abstract meaning of a sequential program is a *function*, from some data to other data. In order to simplify our discussion, let us assume that data belong to a single set S of *objects* (for instance $s \in S$ could be the state of a memory where data are stored). Then the interpretation of a sequential program p is a mapping $\lceil p \rceil : S \to S$.

Ignoring most of the syntax of sequential programs, we may at least suppose that one can form the composition $p;q$. More precisely, let us assume that we have a *monoid* of (sequential) programs: there exists an "empty" program $\mathbb{1}$—meaning termination—, and the laws A0 and U0 are satisfied. Then the interpretation of a program $p;q$ is the functional composition of the interpretations of its components, and the interpretation of $\mathbb{1}$ is identity:

(i) $\lceil \mathbb{1} \rceil = \mathrm{id}_S$,

(ii) $\lceil p;q \rceil = \lceil q \rceil \circ \lceil p \rceil$.

Mathematically speaking, this is just to say that the interpretation $\lceil . \rceil$ is an *operation* of the monoid of programs on the set S of objects.

Let us say a few words about the *operational* definition of the semantics of sequential programs. At a very low level this is described using some kind of *abstract machine*—which could be something like Landin's SECD machine, or the SMC machine (cf. [42, 53]). Let us climb one step in level of abstraction, and say that the operational semantics is given by an unlabelled transition relation on *systems* (=programs+data, cf. [53])

$$(\textit{program}, \textit{object}) \twoheadrightarrow (\textit{program}', \textit{object}').$$

Denoting by $\twoheadrightarrow^*$ the reflexive and transitive closure of this relation, one has the

following operational foundation for the semantics (cf. [42, 53]):

$$\lceil p \rceil(s) = s' \Leftrightarrow (p, s) \xrightarrow{*}\!\!\!\!\rightarrow (\mathbb{1}, s').$$

Note that $\lceil p \rceil(s) = s'$ holds by virtue of a *terminated sequence* of elementary steps (the configuration $(\mathbb{1}, s')$ is a terminal one).

We now aim at defining the operation of concurrent and nondeterministic programs, trying to generalize what we just said about sequential programs. The first point is that these programs cannot be interpreted as functions. We cannot imagine at this moment what could be an abstract mathematical model for the operation of such programs, so let us stick to the concrete level, that of states and arrows. Another standard notation for $f(s) = s'$ is $f : s \mapsto s'$ or $s \mapsto^{f} s'$. This suggests that objects are states of a new kind of labelled transition system, where the labels are programs (or, more accurately, interpretations of programs). Then $s \mapsto^{\lceil p \rceil} s'$ means that the program p operates on the object s, changing it into s'. This obviously copes with nondeterminism and partial mappings since we are not compelled to assume that for each s and p there is exactly one such s'. Regarding the properties of operation with respect to sequential composition, we just translate (i) and (ii) above; hence we require:

(i) $\quad s \xmapsto{\lceil \mathbb{1} \rceil} s,$

(ii) $\quad s \xmapsto{\lceil p \rceil} s'' \xmapsto{\lceil q \rceil} s' \Rightarrow s \xmapsto{\lceil p;q \rceil} s'.$

We must also postulate a general property according to which $s \mapsto^{\lceil p \rceil} s'$ if p performs a terminated sequence of computations which transforms s into s'. We have previously formalized the relation "p performs a computation" by means of the execution transition relation $p \rightarrow^{u} q$. Hence we look for a property which implies

$$p \xrightarrow{u_1} \cdots \xrightarrow{u_k} \mathbb{1} \;\&\; s \xmapsto{\lceil u_1 \rceil} \cdots \xmapsto{\lceil u_k \rceil} s' \Rightarrow s \xmapsto{\lceil p \rceil} s'.$$

An appropriate property may be formulated as follows:

(iii) $\quad p \xrightarrow{u} q,\; s \xmapsto{\lceil u;q \rceil} s' \Rightarrow s \xmapsto{\lceil p \rceil} s'.$

We may regard the three properties (i), (ii), (iii) above as axiomatizing the notion of operation of processes—assuming a sequential composition construct and a notion of execution. As a matter of fact, one could define the transition relation $\twoheadrightarrow$ by

$$p \xrightarrow{u} p' \;\&\; s \xmapsto{\lceil u \rceil} s' \Leftrightarrow (p, s) \twoheadrightarrow (p', s')$$

since then we would have, as for sequential programs,

$$s \xmapsto{\lceil p \rceil} s' \Leftrightarrow (p, s) \xrightarrow{*}\!\!\!\!\rightarrow (\mathbb{1}, s').$$

This shows that the transitions of a configuration (p, s) have two complementary parts, resulting from the execution of a computation u, by the program p, and from the operation of this computation on the object s.

It is a standard point of view, advocated by Milner and Hoare, that there should be no difference between processes and objects: both are states of transition systems. Such discrete systems may be used for instance to model idealized circuits, or memory registers or programs. Let us quote Hoare's book on this subject ([32, p. 65] introducing "interaction"): "the environment of a process may be described as a process (...). This permits investigation of the behaviour of a complete system composed from the process together with its environment, acting and interacting with each other as they evolve concurrently. The complete system should also be regarded as a process (...). In fact, it is best to forget the distinction between processes, environments, and systems". In other words, it would be best to forget the distinction between p, s and (p, s)—the latter being just Hoare's concurrent composition $(p \| s)$. However, we shall depart from this view here: from a syntactical point of view, objects and processes will be terms of slightly different algebras. Moreover, as the last section of the paper will show, it is worth maintaining a semantic asymmetry between objects and programs. This asymmetry appears in the semantics of "systems" (p, s), made out of a process p and an object s. As we have seen, the semantics of such a system requires an *execution* (of the process) and an *operation* (on the object): these will be formalized by the two transition relations $\rightarrow$ and $\mapsto$.

Let us see an example of object, describing the *boolean*. We assume that the set A of atoms contains primitive instructions such as assignments of true or false, respectively a_0 and a_1, and read actions c_0 and c_1. Then the boolean, initially undefined, is represented by the transition system shown in Fig. 5. We shall denote this transition system by $\mathbf{B}$.

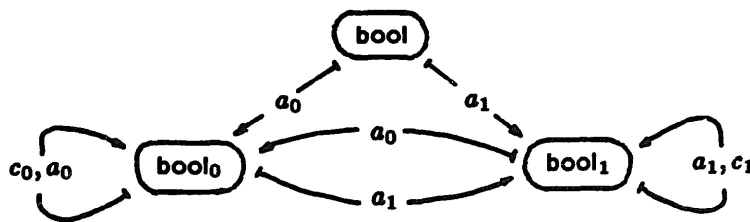


Fig. 5.

6.2. *The algebra of processes*

To build processes we will add two new constructs to those of $\mathbf{T}^{\text{rec}}(A)$. Since processes may operate on objects, we need a construct that combines a process p and an object s to yield a process behaving like the "system" (p, s). More precisely a process is taken to apply to a *named* object: our first new construct, called *block*, is $p\langle l{:}s\rangle$ where p is a process, s an object, and l is a name. Intuitively, $p\langle l{:}s\rangle$ is the process p supplied with a *local object* s named l, what could be $(\text{let } l{:}s \text{ in } p)$ in a

usual notation. It behaves like the system (p, s), but the object s only reacts to the part of the computations of p which is applied to l. In order to specify the part of a computation applied to a named object, we use Hoare's *naming* [32, 9], which we denote $(l.p)$ and call *qualification*. For instance, $(p\|q)\langle l{:}s\rangle$ represents a system of two concurrent processes p and q sharing a common object s, on which they operate by means of computations of the form $l.u$.

To state the syntax of processes, we need, together with the set A of atoms and the set X of identifiers, a denumerable set Λ of *names*; we assume these three sets to be pairwise disjoint. We first define the set $\mathbf{C}_\Lambda(A)$ of *computations* over A.

Computations. (i) $\mathbb{1}$ is a term of $\mathbf{C}_\Lambda(A)$;

(ii) every atom $a \in A$ is a term of $\mathbf{C}_\Lambda(A)$;

(iii) if u and v are terms of $\mathbf{C}_\Lambda(A)$, then $(u;v)$ and $(u\|v)$ are terms of $\mathbf{C}_\Lambda(A)$;

(iv) if $l \in \Lambda$ is a name and u is a term of $\mathbf{C}_\Lambda(A)$, then $(l.u)$ is a term of $\mathbf{C}_\Lambda(A)$.

Our calculus of processes is parametrized on a given *system of objects*. This is a transition system $\Sigma = (Q, \mathbf{C}_\Lambda(A), \sigma)$, where the states $s \in Q$ are objects, and the transition relation $\sigma \subseteq Q \times \mathbf{C}_\Lambda(A) \times Q$ represents the operation of computations on objects. For instance, $\mathbf{B}$ may be a part of such a system, which describes the boolean. We shall see in the next section how to introduce structured objects. For each finite subset Y of X we define the set $\mathbf{P}_\Lambda(A \cup Y, \Sigma)$ of *processes* (with free identifiers in Y). The syntax is as follows.

Processes. (i) $\mathbb{1}$ is a term of $\mathbf{P}_\Lambda(A, \Sigma)$;

(ii) every atom or identifier $y \in A \cup X$ is a term of $\mathbf{P}_\Lambda(A \cup \{y\}, \Sigma)$;

(iii) if p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Sigma)$ and q is a term of $\mathbf{P}_\Lambda(A \cup Y', \Sigma)$, then $(p;q)$, $(p\|q)$ and $(p+q)$ are terms of $\mathbf{P}_\Lambda(A \cup Y \cup Y', \Sigma)$;

(iv) if $x \in X$ is an identifier and p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Sigma)$, then $\mu x.p$ is a term of $\mathbf{P}_\Lambda(A \cup Y - \{x\}, \Sigma)$;

(v) if $l \in \Lambda$ is a name and p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Sigma)$, then $(l.p)$ is a term of $\mathbf{P}_\Lambda(A \cup Y, \Sigma)$;

(vi) if $l \in \Lambda$ is a name, p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Sigma)$ and s is an object, that is, $s \in Q$, then $p\langle l{:}s\rangle$ is a term of $\mathbf{P}_\Lambda(A \cup Y, \Sigma)$.

To state the rules of the operational semantics, we need some auxiliary definitions. First, we must extend the equational theory of termination; the axioms of this theory are U0, U1, U2 and:

U3: $\mu x.\mathbb{1} = \mathbb{1}$,

U4: $(l.\mathbb{1}) = \mathbb{1}$,

U5: $\mathbb{1}\langle l{:}s\rangle = \mathbb{1}$.

For simplicity we shall denote by $\equiv$ the congruence of $\mathbf{P}_\Lambda(A, \Sigma)$ whose axioms are those of Θ, together with U3, U4, U5, and the axioms D1 and D2 that will be given later (the theory of $\equiv$ is $\Theta \cup \Theta'$, cf. Appendix B).

Next we need the notions of *restriction* and *concealment* of computations with respect to names. Let $l \in \Lambda$ be a name; the restriction and concealment of the computation u with regard to l are respectively denoted by $u/_l$ and $u\backslash^l$. These will be used in the semantics of the block construct $p\langle l{:}s\rangle$. Roughly speaking, if p performs a computation u and if $u/_l$ operates on s, then $p\langle l{:}s\rangle$ will perform the computation $u\backslash^l$. But this is not quite correct since $u/_l$ breaks up the causal dependencies that the computation u involves. Let us explain this point: if $u = (l'.v);(l.w)$, then we will have $u/_l \equiv w$, but we cannot regard u as being applicable to an object named l (unless $v \equiv \mathbb{1}$). To be applicable to an object named l, a computation u must be *causally coherent* with respect to l, in notation $u \circledC l$. Intuitively, $u \circledC l$ holds if $u/_l$ is a prefix of u. The formal definitions are as follows:

$$u/_l = \mathbb{1} \quad \text{if } u \in A \cup \{\mathbb{1}\};$$

$$(u;v)/_l = (u/_l;v/_l) \quad \text{and} \quad (u\|v)/_l = (u/_l\|v/_l);$$

$$(l'.u)/_l = \begin{cases} u & \text{if } l' = l, \\ \mathbb{1} & \text{otherwise.} \end{cases}$$

As one can see, $u/_l$ is the part of the computation u which is applied to l. On the other hand, $u\backslash^l$ is u with this part cancelled:

$$u\backslash^l = u \quad \text{if } u \in A \cup \{\mathbb{1}\};$$

$$(u;v)\backslash^l = (u\backslash^l;v\backslash^l) \quad \text{and} \quad (u\|v)\backslash^l = (u\backslash^l\|v\backslash^l);$$

$$(l'.u)\backslash^l = \begin{cases} \mathbb{1} & \text{if } l' = l, \\ (l'.u) & \text{otherwise.} \end{cases}$$

Finally, $\circledC$ is the least relation satisfying the following clauses:

(i) if $u \in A \cup \{\mathbb{1}\}$, then $u \circledC l$;
(ii) $(l'.u) \circledC l$ for all $u \in \mathbf{C}_\Lambda(A)$ and $l' \in \Lambda$;
(iii) if $u \circledC l$ and $v/_l \equiv \mathbb{1}$, then $(u;v) \circledC l$;
(iv) if $u\backslash^l \equiv \mathbb{1}$ and $v \circledC l$, then $(u;v) \circledC l$;
(v) if $u \circledC l$ and $v \circledC l$, then $(u\|v) \circledC l$.

We now define the transition relation $\to$, which is the least subset of $\mathbf{P}_\Lambda(A, \Sigma) \times \mathbf{C}_\Lambda(A) \times \mathbf{P}_\Lambda(A, \Sigma)$ satisfying the rules E (of execution). We shall give some intuitive explanations about these rules later.

E1: *atom*

$$a \in A \vdash a \xrightarrow{a} \mathbb{1},$$

E2.1: *sequential composition* 1

$$p \xrightarrow{u} p' \vdash (p;q) \xrightarrow{u} (p';q),$$

E2.2.1: *sequential composition* 2

$$p \equiv \mathbb{1},\ q \xrightarrow{v} q' \vdash (p;q) \xrightarrow{v} q',$$

E2.2.2: *sequential composition* 3

$$p \xrightarrow{u} p' \equiv \mathbb{1},\ q \xrightarrow{v} q' \vdash (p;q) \xrightarrow{(u;v)} q',$$

E3.1: *parallel composition* 1

$$p \xrightarrow{u} p' \vdash (p \| q) \xrightarrow{u} (p' \| q),$$

E3.2: *parallel composition* 2

$$p \xrightarrow{u} p',\ q \xrightarrow{v} q' \vdash (p \| q) \xrightarrow{(u \| v)} (p' \| q'),$$

E3.3: *parallel composition* 3

$$q \xrightarrow{v} q' \vdash (p \| q) \xrightarrow{v} (p \| q'),$$

E4.1: *sum* 1

$$p \xrightarrow{u} p' \vdash (p+q) \xrightarrow{u} p',$$

E4.2: *sum* 2

$$q \xrightarrow{v} q' \vdash (p+q) \xrightarrow{v} q',$$

E5: *fixpoint*

$$p[\mu x.p/x] \xrightarrow{u} p' \vdash \mu x.p \xrightarrow{u} p',$$

E6: *qualification*

$$p \xrightarrow{u} p' \vdash (l.p) \xrightarrow{(l.u)} (l.p'),$$

E7: *block*

$$p \xrightarrow{u} p' \ \&\ u \circledcirc l,\ (s, u/_l, s') \in \sigma \vdash p\langle l{:}s\rangle \xrightarrow{u\backslash^l} p'\langle l{:}s'\rangle.$$

Let us make some comments about these execution rules: first of all, there is no rule for $\mathbb{1}$; for instance, we cannot prove $\mathbb{1} \to^{\mathbb{1}} \mathbb{1}$, nor, more generally, $p \to^{\mathbb{1}} p$. This explains why we have split the rules for sequential and parallel composition. On the other hand, rule E7 may introduce $\mathbb{1}$ as a computation, in a transition $p \to^{\mathbb{1}} p'$ (if $u\backslash^l = \mathbb{1}$). This is an "internal move", which is a meaningful step. Therefore, we do not exclude the cases $u \equiv \mathbb{1}$ and $v \equiv \mathbb{1}$ in the rules E4.1 and E4.2 for the sum, while they ought to be precluded if $p \to^{\mathbb{1}} p$ were an axiom.

Let us comment on the rule E7 for $q = p\langle l{:}s\rangle$. We have said that such a process could be written (let $l{:}s$ in p). Then q behaves as the process p operating on a *local* object whose name is l, and whose state is s. The block operator is asymmetric in two respects: the execution of q requires an execution of p and an operation on

s—this appears in the hypothesis of E7. Moreover, it behaves as a "left-merge with synchronization" since $p\langle l, s\rangle$ cannot perform anything unless the process p—the active component of $p\langle l:s\rangle$—performs some computation u. The part of this computation u which is applied to l, that is, $u/_l$, operates on s—the object s is the passive component of $p\langle l:s\rangle$. The computation $u/_l$ is consumed during the operation, and is thus concealed form the resulting computation of q, which is then $u\backslash^l$. Concealment expresses the local character of the name l. Then an "internal move" arises when the whole computation u of p is applied to the local object, that is, when $u\backslash^l \equiv \mathbb{1}$—or if p itself performs such a silent transition. We must emphasize the fact that the block construct is the *only synchronization and communication mechanism* of our calculus. This communication is similar to *application* of functional languages since it consists in "applying" to some object an elementary computation of a process, at each step of its execution.

Let us see an example, showing how to model a boolean conditional. We assume that a_0, a_1, c_0 and c_1 are primitive instructions, belonging to A, and that $\mathbf{B}$ is (part of) the given system of objects. Let $r = ((b.c_0);p + (b.c_1);q)$. Then

$$(r\|b.a_0)\langle b:\mathsf{bool}\rangle \xrightarrow{\mathbb{1}} (r\|b.\mathbb{1})\langle b:\mathsf{bool}_0\rangle \quad \text{by E1, E6, E3.3, and E7,}$$

$$(r\|b.\mathbb{1})\langle b:\mathsf{bool}_0\rangle \xrightarrow{\mathbb{1}} (p\|b.\mathbb{1})\langle b:\mathsf{bool}_0\rangle \quad \text{by E1, E6, E4.1, and E7.}$$

One can remark that the term $(p\|b.\mathbb{1})\langle b:\mathsf{bool}_0\rangle$ has a behaviour similar to that of $p\langle b:\mathsf{bool}_0\rangle$ since $(b.\mathbb{1})$ cannot perform anything. One can use the abbreviations:

$$\begin{aligned}
(\textsf{if } b \textsf{ then } p \textsf{ else } q) \quad &\text{for} \quad ((b.c_0);p + (b.c_1);q),\\
(\textsf{while } b \textsf{ do } p) \quad &\text{for} \quad \mu x.((b.c_0);p;x + (b.c_1)),\\
(\textsf{when } b \textsf{ do } p) \quad &\text{for} \quad ((b.c_0);p)\\
&\text{or} \quad \mu x.((b.c_0);p + (b.c_1);x).
\end{aligned}$$

To conclude this section, let us say a few words about equipollence of processes. Once more, we wish to deal with "semantic" transitions, that is, transitions labelled by equivalence classes of computations. It should be clear that computations of $\mathbf{C}_\Lambda(A)$ denote finite posets labelled by *qualified atoms*. These qualified atoms are the terms built using only the formation rules (ii) and (iv) of $\mathbf{C}_\Lambda(A)$. Then $(l.u)$ denotes the same structure as u but with every label prefixed by l. Therefore, the axioms of computation equality are those of Δ, plus U4 and

D1: $(l.(p;q)) = ((l.p);(l.q))$,

D2: $(l.(p\|q)) = ((l.p)\|(l.q))$.

(these axioms belong to Θ', see Appendix B). Then the equipollence $\asymp$ is defined with respect to the equivalence of computations, namely $\equiv$. Recall that $\asymp$ is the coarsest equivalence satisfying

$$p \asymp q \;\&\; p \xrightarrow{u} p' \Rightarrow \exists v \equiv u \;\exists q' \asymp p' \; q \xrightarrow{v} q'.$$

For example, we have the distributivity of the conditional branching over sequential composition:

$$(\text{if } b \text{ then } p \text{ else } q);r \asymp (\text{if } b \text{ then } p;r \text{ else } q;r).$$

We must point out that among the axioms of $\equiv$ for computations, one has U0 and U1, that is,

$$(u;\mathbb{1}) = u = (\mathbb{1};u), \qquad (u\|\mathbb{1}) = u = (\mathbb{1}\|v).$$

Therefore, $\mathbb{1}$ shows some analogy with the τ of CCS, and even more with the unit action of MEIJE/SCCS. But $\asymp$ is not an observational equivalence since, for instance, $p \to^{u} p'' \to^{v} p'$ cannot be confused with $p \to^{u;v} p'$. The equipollence is still a congruence with respect to the operators since it is defined as a strong bisimulation.

7. Objects, abstraction and communication

7.1. Objects and atomic actions

In this section we introduce a syntax for objects, including a construct for defining atomic actions, and we formalize the operation of processes on objects. This will give us the system of objects $\Sigma = (Q, \mathbf{C}_\Lambda(A), \sigma)$ that was used as a parameter for the algebra of processes in the previous section.

The algebra of objects is itself parameterized on a system of *primitive* objects, which could be the provided data and instructions of an abstract machine. Then we assume the set A of atoms to be the union of two disjoint sets I and Z: I is a nonempty set of *primitive instructions*, and Z is a denumerable set of *atom identifiers*. Let us denote by $I^{\circledcirc}$ the least subset of $\mathbf{C}_\Lambda(A)$ containing I, and such that $u, v \in I^{\circledcirc} \Rightarrow (u\|v) \in I^{\circledcirc}$. We take the system of primitive objects to be a transition system $\Xi = (O, I^{\circledcirc}, \xi)$. The transition relation $\xi \subseteq O \times I^{\circledcirc} \times O$ gives us the operation of primitive instructions over primitive objects. Intuitively, the constants $o \in O$ are interpreted as *values*, and thus we postulate that if $(o, u, o') \in \xi$, then u does not have the form $(l.v)$. On the other hand, u cannot be $(v;w)$: this *interruptibility* property means that the grain of atomicity cannot be finer for processes than for primitive objects. Moreover, we shall say that a system Ξ satisfies a *mutual exclusion* property for the primitive instructions if it satisfies $\xi \subseteq O \times I \times O$.

Objects share some constructors with processes, namely sum, parallel composition, and fixpoint. The atoms $a \in A$ and the unit $\mathbb{1}$ are not allowed as objects, and one cannot use sequential composition nor block to build objects. On the other hand, the *abstraction construct* $\{\alpha z_1, \ldots, z_k.p_1, \ldots, p_k\}s$ is specific to objects. Corresponding to qualification $(l.p)$ for processes, we have for objects a *declaration* construct $(l::s)$, where l is a name and s an object. Intuitively, $(l::s)$ is "an object named l whose state is s".

The algebra of objects, based upon a system Ξ of primitive objects, is denoted $\mathbf{U}_\Lambda(A, \Xi)$. More precisely, we define for any finite set Y of identifiers (subset of

X) the set of terms $\mathbf{U}_\Lambda(A \cup Y, \Xi)$. The set $\mathbf{U}_\Lambda(A, \Xi)$ of closed terms is the set of objects used to build processes. For simplicity, the algebra of processes will be denoted $\mathbf{P}_\Lambda(A, \Xi)$, instead of $\mathbf{P}_\Lambda(A, (\mathbf{U}_\Lambda(A, \Xi), \mathbf{C}_\Lambda(A), \theta))$. We give the whole syntax to avoid misunderstanding.

Objects. (i) every identifier $y \in X$ is a term of $\mathbf{U}_\Lambda(A \cup \{y\}, \Xi)$. Every constant $o \in O$ is a term of $\mathbf{U}_\Lambda(A, \Xi)$;

(ii) if s is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$ and r is a term of $\mathbf{U}_\Lambda(A \cup Y', \Xi)$, then $(s \| r)$ and $(s+r)$ are terms of $\mathbf{U}_\Lambda(A \cup Y \cup Y', \Xi)$;

(iii) if $x \in X$ is an identifier and s is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$, then $\mu x.s$ is a term of $\mathbf{U}_\Lambda(A \cup Y - \{x\}, \Xi)$;

(iv) if $l \in \Lambda$ is a name and s is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$, then $(l{::}s)$ is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$;

(v) if $z_1, \ldots, z_k \in Z$ are distinct atom identifiers, $p_1, \ldots, p_k$ are terms of $\mathbf{P}_\Lambda(A, \Xi)$, and s is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$, then $\{\alpha z_1, \ldots, z_k.p_1, \ldots, p_k\}s$ is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$.

Processes. (i) $\mathbb{1}$ is a term of $\mathbf{P}_\Lambda(A, \Xi)$;

(ii) every atom or identifier $y \in A \cup X$ is a term of $\mathbf{P}_\Lambda(A \cup \{y\}, \Xi)$;

(iii) if p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$ and q is a term of $\mathbf{P}_\Lambda(A \cup Y', \Xi)$, then $(p;q)$, $(p \| q)$ and $(p+q)$ are terms of $\mathbf{P}_\Lambda(A \cup Y \cup Y', \Xi)$;

(iv) if $x \in X$ is an identifier and p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$, then $\mu x.p$ is a term of $\mathbf{P}_\Lambda(A \cup Y - \{x\}, \Xi)$;

(v) if $l \in \Lambda$ is a name and p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$, then $(l.p)$ is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$;

(vi) if $l \in \Lambda$ is a name, p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$ and s is a term of $\mathbf{U}_\Lambda(A, \Xi)$, then $p\langle l{:}s\rangle$ is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$.

We shall use the more compact notation $\{\alpha\bar{z}.\bar{p}\}s$ for $\{\alpha z_1, \ldots, z_k.p_1, \ldots, p_k\}s$. In a concrete syntax, this term could be written:

$$(s \textsf{ with } z_1 = p_1, \ldots, z_k = p_k).$$

This is to point out the similarity between our abstraction construct and that of abstract data types. The concept of abstract data type is well-known: roughly speaking an ADT is a module, regarded as the manager of objects of some class, offering a collection of procedures to alter the state of the object. Moreover, it is intended that the details of implementing the objects and procedures are hidden into the module—or equivalently that programs using the module only know its specification. Accordingly, the informal meaning of $\{\alpha\bar{z}.\bar{p}\}s$ is the following: s is the internal state of this object, the $z_1, \ldots, z_k$ are the names of the available procedures and the processes $p_1, \ldots, p_k$ are their respective codes. The term $\{\alpha\bar{z}.\bar{p}\}s$ is intended to only react to computations built with the $z_1, \ldots, z_k$, and more precisely to computations of $\mathbf{D}(\{z_1, \ldots, z_k\})$ (see Appendix A), built without qualification.

This will appear in the rules of operations, which we shall describe next. We must note however that the execution of an atom $z \in Z$ is *indivisible* since we always have, by virtue of E1, $\forall z \in Z \xrightarrow{z} \mathbb{1}$. Regarded as a process, an identifier $z \in Z$ is an *atom call*.

We will use $\lceil p \rceil$ as a notational trick, suggested by Berry, to distinguish the (closed) process p from its operation. We call such a $\lceil p \rceil$ an *operation.*

Operations. (i) If p is a term of $\mathbf{P}_\Lambda(A, \Xi)$, then $\lceil p \rceil$ is an operation, belonging to $\mathbf{O}_\Lambda(A, \Xi)$.

Then $\mapsto$ is the least subset of $\mathbf{U}_\Lambda(A, \Xi) \times \mathbf{O}_\Lambda(A, \Xi) \times \mathbf{U}_\Lambda(A, \Xi)$ satisfying the rules O (of operation) below. It must be understood that the operation relation σ used in rule E7 for processes of $\mathbf{P}_\Lambda(A, \Xi)$ is given by

$$\sigma \overset{\text{def}}{=} \mapsto \cap (\mathbf{U}_\Lambda(A, \Xi) \times \mathbf{C}_\Lambda(A) \times \mathbf{U}_\Lambda(A, \Xi)).$$

In other words, the hypothesis $(s, u/_l, s') \in \sigma$ of rule E7 should be replaced by $s \mapsto^{\lceil u/_l \rceil} s'$.

There are three kinds of rules for the operation: first of all, the axioms are the transitions of the given system of primitive objects, that is, $(o, u, o') \in \xi$. Moreover, the operation $\mapsto$ satisfies some specific properties, as we have seen in Section 6.1. These are expressed by the rules O1 and O2 below, which, as we shall see, ensure that processes operate on objects by means of their terminated sequences of computations. Finally, there are the structural rules O3 to O7, which are similar to those of execution. In the rule for $\{\alpha \bar{z}.\bar{p}\}s$, the *abstraction* rule O7, we use the following convention: $\bar{z} = z_1, \ldots, z_k$ and $\bar{p} = p_1, \ldots, p_k$. As before $u[\bar{p}/\bar{z}]$ denotes the substitution of the p_i's for the z_i's in u.

O0: *reaction*

$$(o, u, o') \in \xi \vdash o \xmapsto{\lceil u \rceil} o',$$

O1: *identity*

$$p \equiv \mathbb{1} \vdash s \xmapsto{\lceil p \rceil} s,$$

O2: *operation*

$$p \xrightarrow{u} p',\ s \xmapsto{\lceil u \rceil} s'' \xmapsto{\lceil p' \rceil} s' \vdash s \xmapsto{\lceil p \rceil} s',$$

O3.1: *parallel composition* 1

$$s \xmapsto{\lceil u \rceil} s' \vdash (s \| r) \xmapsto{\lceil u \rceil} (s' \| r),$$

O3.2: *parallel composition* 2

$$r \xmapsto{\lceil v \rceil} r' \vdash (s \| r) \xmapsto{\lceil v \rceil} (s \| r'),$$

O4.1: *sum* 1

$$s \xmapsto{\lceil u \rceil} s' \;\&\; u \not\equiv \mathbb{1} \vdash (s+r) \xmapsto{\lceil u \rceil} s',$$

O4.2: *sum* 2

$$r \xmapsto{\lceil v \rceil} r' \;\&\; v \not\equiv \mathbb{1} \vdash (s+r) \xmapsto{\lceil v \rceil} r',$$

O5: *fixpoint*

$$s[\mu x.s/x] \xmapsto{\lceil u \rceil} s' \vdash \mu x.s \xmapsto{\lceil u \rceil} s',$$

O6: *declaration*

$$s \xmapsto{\lceil u \rceil} s' \vdash (l.s) \xmapsto{[l::u]} (l.s'),$$

O7: *abstraction*

$$s \xmapsto{\lceil u[\bar{p}/\bar{z}] \rceil} s',\; u \in \mathbf{D}(\{\bar{z}\}) \vdash \{\alpha\bar{z}.\bar{p}\}s \xmapsto{\lceil u \rceil} \{\alpha\bar{z}.\bar{p}\}s'.$$

Note. There is some implicit typing in these rules: in O0 and in O3 to O6, u and v are computations of $\mathbf{C}_\Lambda(A)$.

A few remarks about these rules: a first one is that an object cannot make an autonomous silent transition, for $s \mapsto^{\lceil p \rceil} s' \;\&\; p \equiv \mathbb{1} \Rightarrow s' = s$. Then an object is a "passive" agent. There is no rule similar to E3.2, and the operation of a parallel process $(p \| q)$ must be introduced by O0, O1, or O2. On the other hand, the rules O3 state some inheritance phenomena: if a part of such a compound object reacts to some computation, then the whole object accepts the same operation. In the abstraction rule, the hypothesis $u \in \mathbf{D}(\{\bar{z}\})$ expresses the fact that a process has only access to the abstract object $\{\alpha\bar{z}.\bar{p}\}s$ by means of its specified interface. Stated equivalently, a process cannot directly manipulate the internal structure of an object.

The following result asserts that the O rules properly define $\mapsto$ with regard to the idea that a program operates by means of its terminated sequences of computations.

Proposition 7.1. *Let* $p \twoheadrightarrow p' \Leftrightarrow^{\text{def}} \exists u \equiv \mathbb{1}\; p \rightarrow^{u} p'$. *Then for all processes* $p \in \mathbf{P}_\Lambda(A, \Xi)$ *and for all objects* $s, s' \in \mathbf{U}_\Lambda(I, \Xi)$ *the following properties are equivalent*:

(i) $s \xmapsto{\lceil p \rceil} s'$,

(ii) $\exists n\, \exists u_1, \ldots, u_n\, \exists p' \equiv \mathbb{1} \quad p \xrightarrow{u_1} \cdots \xrightarrow{u_n} p' \;\&\; s \xmapsto{\lceil u_1 \rceil} \cdots \xmapsto{\lceil u_n \rceil} s'$,

(iii) $\exists p' \equiv \mathbb{1} \quad (l.p)\langle l:s \rangle \overset{*}{\twoheadrightarrow} (l.p')\langle l:s' \rangle$.

The (easy) proof is omitted.

One can regard the equivalence (i)⇔(iii) as a generalization of the relationship between the denotational and operational semantics for sequential programs: the transitions $s \mapsto^{\lceil p \rceil} s'$ define the "abstract" operation of p, while $(l.p)\langle l{:}s\rangle$ is a "configuration" which may evolve step by step to a terminal one where $p \equiv \mathbb{1}$. On the other hand, the equivalence (i)⇔(ii) entails a *recoverability* property for our atomic actions (cf. [38, 10]), which operate in an all-or-nothing manner. Let us explain this point better: in order to prove $\{\alpha\bar{z}.\bar{p}\}s \mapsto^{\lceil u \rceil} \{\alpha\bar{z}.\bar{p}\}s'$, one has to prove $s \mapsto^{\lceil u[\bar{p}/\bar{z}] \rceil} s'$. This holds if

$$\exists n > 0\ \exists v_1, \ldots, v_n \in \mathbf{C}_\Lambda(A) \quad u[\bar{p}/\bar{z}] \xrightarrow{v_1} \cdots \xrightarrow{v_n} p' \equiv \mathbb{1}\ \&\ s \overset{\lceil v_1 \rceil}{\longmapsto} \cdots \overset{\lceil v_n \rceil}{\longmapsto} s'.$$

But since u is a finite, determinate computation, each code p_i occurring in the term $u[\bar{p}/\bar{z}]$ must contribute to the terminated execution of this term, and this contribution is a terminated sequence of computations of p_i.

Usually one assumes not only recoverability for atomic actions, but also *serializability* (or noninterference, cf. [38, 10]). This property states that atomic actions operate as if they were mutually exclusive. We could have accounted for such a property by assuming another rule for abstraction, namely

$$s \overset{\lceil p_i \rceil}{\longmapsto} s' \vdash \{\alpha\bar{z}.\bar{p}\}s \overset{\lceil z_i \rceil}{\longmapsto} \{\alpha\bar{z}.\bar{p}\}s'.$$

Then the abstraction construct $\{\alpha\bar{z}.\bar{p}\}s$ would actually define a *monitor*, where the procedures are mutually exclusive. The previous restricted rule enforces a strict exclusion; then it does not allow, for instance, an interleaved operation of concurrent transactions, where serializability is a criterion for data consistency. We shall not adopt this restricted rule here, for we want to be able to deal with mutual inclusion (rendez-vous), as well as mutual exclusion, by means of atomic actions. Moreover, it it is easier to understand serializability as an algorithmic problem rather than a requisite for a compiler—or a formal semantics: one can write the code of atoms for an abstract object in such a way that they satisfy serializability, without changing the semantics.

Another interesting consequence of the previous result is the following corollary.

Corollary 7.2. *For all processes p, $q \in \mathbf{P}_\Lambda(A, \Xi)$ and for all objects s, $s' \in \mathbf{U}_\Lambda(I, \Xi)$ we have the following sequentialization properties*:

$$\exists s'' \quad s \overset{\lceil p \rceil}{\longmapsto} s'' \overset{\lceil q \rceil}{\longmapsto} s' \Leftrightarrow s \overset{\lceil p;q \rceil}{\longmapsto} s' \Rightarrow s \overset{\lceil p \| q \rceil}{\longmapsto} s'.$$

Moreover, we have the derived rule

$$\text{Opar:} \quad s \overset{\lceil p \rceil}{\longmapsto} s',\ r \overset{\lceil q \rceil}{\longmapsto} r' \Rightarrow (s \| r) \overset{\lceil p \| q \rceil}{\longmapsto} (s' \| r').$$

(A proof would use characterization (ii) of operation in the previous proposition, and the E rules for sequential and parallel composition to build the suitable sequence of executions.)

Let us give a simple example of abstract object, that of a *boolean semaphore*. For the rest of the paper, the only given object is the boolean—that is, our algebras of objects and processes are $\mathbf{U}_A(A, \mathbf{B})$ and $\mathbf{P}_A(A, \mathbf{B})$. The semaphore is the following abstract object of $\mathbf{U}_A(A, \mathbf{B})$:

$$\mathsf{sem} = ((b{::}\mathsf{bool}_0) \text{ with } P = (b.c_0);(b.a_1),\ V = (b.c_1);(b.a_0)).$$

According to our previous conventions, P is (when b do $(b.a_1)$), whereas V is (when $\neg b$ do $(b.a_0)$). We clearly have

$$(sem)\quad \mathsf{sem} \xmapsto{\lceil P \rceil} \mathsf{sem}' \xmapsto{\lceil V \rceil} \mathsf{sem}$$

(where sem' is the same as sem but with b in state bool_1) since

$$\mathsf{bool}_0 \xmapsto{\lceil (b.c_0);(b.a_1) \rceil} \mathsf{bool}_1 \xmapsto{\lceil (b.c_1);(b.a_0) \rceil} \mathsf{bool}_0.$$

Then we will have $\mathsf{sem} \mapsto^{\lceil P;V \rceil} \mathsf{sem}$, but also $\mathsf{sem} \mapsto^{\lceil P \| V \rceil} \mathsf{sem}$ (applying the previous corollary). This shows that in a construct $p\langle l{:}\mathsf{sem}\rangle$, the execution of the process p is not much constrained; what is precluded is, for instance, a computation $V;P$. We could prove that the possible operations $\mathsf{sem} \mapsto^{\lceil p \rceil} s$ are exactly those generated (using the rules O1 and O2) by the transitions

$$\mathsf{sem} \xmapsto{\lceil P \rceil} \mathsf{sem}', \qquad \mathsf{sem}' \xmapsto{\lceil V \rceil} \mathsf{sem}.$$

In other words, the formula (sem) above is a *specification* of the abstract object semaphore, and sem is a correct implementation.

As usual, the semaphore may be used to program *critical regions*, enclosing mutually exclusive pieces of code, as in

$$(\ldots(s.P);p;(s.V)\ldots\|\ldots(s.P);q;(s.V)\ldots)\langle s{:}\mathsf{sem}\rangle$$

for instance. We can use the semaphore to show the necessity of the hypothesis $u \copyright l$ in rule E7, by means of the following example of a *causality cycle*, due to Gonthier: let p be a process performing the computation $p \rightarrow^{u} p'$, where

$$u = ((s.V);(s'.P)\|(s'.V);(s.P)).$$

Then, without the hypothesis $u \copyright l$ in rule E7, we would have

$$p\langle s{:}\mathsf{sem}\rangle\langle s'{:}\mathsf{sem}\rangle \xrightarrow{\mathbb{1}} p'\langle s{:}\mathsf{sem}\rangle\langle s'{:}\mathsf{sem}\rangle$$

since $u/_s \equiv (V\|P)$ and $(u\backslash^s)/_{s'} \equiv (P\|V)$. But this is intuitively unacceptable since the term $u\langle s{:}\mathsf{sem}\rangle\langle s'{:}\mathsf{sem}\rangle$ must be deadlocked.

To conclude this section, let us return for a while to the equivalence of processes. We may call $\asymp$ the *intensional equivalence* since it relies upon the way of computing. On the other hand, the *extensional equivalence* $\approx$ simply means "to operate in the same manner":

$$p \approx q \overset{\text{def}}{\Leftrightarrow} \forall s'\ s \xmapsto{\lceil p \rceil} s' \Leftrightarrow s \xmapsto{\lceil q \rceil} s'.$$

However, this terminology is perhaps not quite right since intensional equivalence does not imply extensional equivalence. The problem is with termination and divergence: for instance, the process $\Omega = \mu x.x$ does not have any execution, therefore $\Omega \asymp \mathbb{1}$; but it is not terminated: $\Omega \neq \mathbb{1}$, and then Ω does not have any operation; consequently, $\Omega \not\approx \mathbb{1}$. We do have $u \asymp v \Leftrightarrow u \approx v$ (and $u \asymp v \Leftrightarrow u \equiv v$) for computations u, v (the proof is left to the reader).

7.2. Synchronization and communication structures

The early interest in concurrent processes came from operating systems (cf. [21]). There the main synchronization problem was to ensure mutually exclusive accesses to shared resources. It is fairly easy to propose an abstract solution to this problem in our formalism, by means of an idealized monitor: let us assume that we want to enforce the mutual exclusion of procedures $p_1, \ldots, p_k$ operating on a shared object o. To this end we define an abstract object t encapsulating o together with a local semaphore and offering the given procedures as new atoms:

$$t = ((l::o \| s::\mathsf{sem}) \text{ with } z_1 = (s.P);(l.p_1);(s.V) \\ \vdots \\ z_k = (s.P);(l.p_k);(s.V)).$$

It should be clear that if $o \mapsto^{[p_i]} o'$, then we have $t \mapsto^{[z_i]} t'$ where

$$t' = \{\alpha\bar{z}.\bar{q}\}(l::o' \| s::\mathsf{sem})$$

(with $q_i = (s.P);(l.p_i);(s.V)$). Therefore, t accepts any computation of $\mathbf{D}(\{z_1, \ldots, z_k\})$, and reacts exactly as if the p_i's were its primitive instructions.

This example suggests that a synchronization or communication structure should be seen as an abstract object $o = \{\alpha\bar{z}.\bar{q}\}s$ where s stores the local data, and where the procedures or methods q_i sharing these data enforce some synchronization or communication discipline. Then a typical system of concurrent processes using such a structure has the form $(r_1 \| \ldots \| r_n)\langle l:o\rangle$. Let us give another example of synchronization structure; we said in Section 3 that the event structure $a\#b\#c\#d$ cannot be the interpretation of a term of $\mathbf{T}(A)$. However, we can define an abstract object whose operations are exactly the computations of this event structure, namely:

$$((s_1::\mathsf{sem} \| s_2::\mathsf{sem} \| s_3::\mathsf{sem}) \text{ with } a = (s_1.P) \\ b = (s_1.P \| s_2.P) \\ c = (s_2.P \| s_3.P) \\ d = (s_3.P)).$$

One should note the formal analogy between this "implementation" and the Petri net associated with the event structure $a\#b\#c\#d$ (cf. Section 3): each semaphore is an input place.

An interesting synchronization problem is that of synchronous message passing in distributed systems. More specifically, let us concentrate on CCS communication. In CCS, an agent possesses some named *ports* through which it may communicate (cf. [43, 44]). In our view, a port is an abstract object offering two atoms for sending and receiving—we shall only deal with pure CCS, without value passing. We aim at defining the port as a communication structure in such a way that the CCS *restriction* operator $p\backslash l$ (or, more accurately, a generalization of this operator) can be defined as:

$$p\backslash l = p\langle l{:}\mathsf{port}\rangle.$$

The definition of the port, using two boolean semaphores, is inspired by [33]. The port is the following abstract object of $\mathbf{U}_{\Lambda}(A, \mathbf{B})$:

$$\mathsf{port} = ((s{::}\mathsf{sem} \| s'{::}\mathsf{sem}) \text{ with } \mathit{send} = (s.P);(s'.V)$$
$$\mathit{receive} = (s'.P);(s.V)).$$

In order to see in detail how abstraction works, we proceed to a complete analysis of the transition

$$r = ((l.\mathit{send});p \| (l.\mathit{receive});q)\langle l{:}\mathsf{port}\rangle \xrightarrow{\mathbb{1}} r' \equiv (p \| q)\langle l{:}\mathsf{port}\rangle \tag{2}$$

assuming the previous specification for the semaphore. In CCS the term r would be written $((l!).p|(l?).q)\backslash l$. We shall use the abbreviations $l!$ for $(l.\mathit{send})$ and $l?$ for $(l.\mathit{receive})$. First of all we have

$$\text{E3.2:}\dfrac{\text{E2.1:}\dfrac{\text{E6:}\dfrac{\text{E1:}\dfrac{\mathit{send} \in A}{\mathit{send} \xrightarrow{\mathit{send}} \mathbb{1}}}{l! \xrightarrow{l!} \mathbb{1}}}{l!;p \xrightarrow{l!} p' = (l.\mathbb{1});p} \qquad \text{E2.1:}\dfrac{\text{E6:}\dfrac{\text{E1:}\dfrac{\mathit{receive} \in A}{\mathit{receive} \xrightarrow{\mathit{receive}} \mathbb{1}}}{l? \xrightarrow{l?} \mathbb{1}}}{l?;q \xrightarrow{l?} q' = (l.\mathbb{1});q}}{(l!;p \| l?;q) \xrightarrow{l!\|l?} (p' \| q')}$$

Since $(l.\mathbb{1}) \equiv \mathbb{1}$ and $\mathbb{1};p \equiv p$, we have $p' \equiv p$ and, similarly, $q' \equiv q$. Clearly, we have $(l!\|l?)$ © l, $(l!\|l?)/_l = (\mathit{send} \| \mathit{receive})$ and $(l!\|l?)\backslash^l = \mathbb{1}$. Then (2) will be proved, using E7, from

$$(l!;p \| l?;q) \xrightarrow{l!\|l?} (p' \| q') \tag{3}$$

which we have just shown, and from

$$\mathsf{port} \xmapsto{[\mathit{send} \| \mathit{receive}]} \mathsf{port}. \tag{4}$$

Let us now prove this second fact. A first step is

$$\text{Opar:}\ \dfrac{\text{O6:}\ \dfrac{\mathsf{sem} \overset{[P]}{\longmapsto} \mathsf{sem}'}{(s{::}\mathsf{sem}) \overset{[s.P]}{\longmapsto} (s{::}\mathsf{sem}')} \qquad \text{O6:}\ \dfrac{\mathsf{sem} \overset{[P]}{\longmapsto} \mathsf{sem}'}{(s'{::}\mathsf{sem}) \overset{[s'.P]}{\longmapsto} (s'{::}\mathsf{sem}')}}{(s{::}\mathsf{sem} \| s'{::}\mathsf{sem} \overset{[s.P\|s'.P]}{\longmapsto} (s{::}\mathsf{sem}' \| s'{::}\mathsf{sem}')}$$

An entirely similar proof would show that

$$(s{::}\mathsf{sem}' \| s'{::}\mathsf{sem}') \overset{[s.V\|s'.V]}{\longmapsto} (s{::}\mathsf{sem} \| s'{::}\mathsf{sem}).$$

We leave it to the reader to prove that

$$(s.P;s'.V \| s'.P;s.V) \overset{s.P\|s'.P}{\longmapsto} ((s.\mathbb{1});s'.V \| (s'.\mathbb{1});s.V) \overset{s.V\|s'.V}{\longmapsto} (\mathbb{1}\|\mathbb{1}).$$

Then, by O2;

$$(s{::}\mathsf{sem} \| s'{::}\mathsf{sem}) \overset{[s.P;s'.V\|s'.P;s.V]}{\longmapsto} (s{::}\mathsf{sem} \| s'{::}\mathsf{sem})$$

Finally, we obtain formula (4) above by applying the abstraction rule.

To prove (2) we have to show that a sequentialization of $(s.P;s'.V\|s'.P;s.V)$ operates on the internal structure of the object port—this sequentialization is $(s.P\|s'.P);(s'.V\|s.V)$, but we could have chosen any stronger sequentialization. On the other hand, it should be clear that one cannot prove that a *send* or a *receive* alone operates on the port since their codes cannot terminate without the cooperation of the other. We could say that the *specification* of the port is

$$(\mathit{port})\quad \mathsf{port} \overset{[\mathit{send}\|\mathit{receive}]}{\longmapsto} \mathsf{port}$$

since this axiom generates exactly, by means of rules O1 and O2, the possible operations on the abstract object port. We may regard the construct $p\langle l{:}\mathsf{port}\rangle$ as defining a restriction operator (the CCS τ internal action being replaced by $\mathbb{1}$). This shows that the *hiding* concept of models of concurrency such as CCS or TCSP is related to the more conventional notion of *scope*.

Generalizing the mutual inclusion problem, we can define a synchronization structure where some given procedures $p_1, \ldots, p_k$, operating on a shared object o, are forced to act *simultaneously*:

$$\begin{aligned}((l{::}o\|(s_1{::}\mathsf{sem}\|\ldots\|s_k{::}\mathsf{sem}))\ \text{with}\ z_1 &= (s_1.P);(l.p_1);(s_2.V)\\ z_2 &= (s_2.P);(l.p_2);(s_3.V)\\ &\vdots\\ z_k &= (s_k.P);(l.p_k);(s_1.V)).\end{aligned}$$

In this way, we are able to organize a rendezvous between a given number of processes; we can also mix the mutual exclusion and mutual inclusion synchroniz-

ation disciplines. Some other examples, as, for instance, the *signal/wait* primitives related to a broadcast event, are given in [7].

8. Conclusion

Summing up our proposal for a calculus of processes, we could say that its three main features are *asynchrony, applicative communication* and *abstraction.* The calculus of processes $\mathbf{P}_\Lambda(A, \Xi)$ that we have proposed owes its asynchronous character mostly to the introduction of structure in the computations. It is evident that our interpretation of parallel composition is asynchronous: as in MEIJE, we think of concurrent processes as independent agents. The rôle of sequential composition is more hidden, but certainly not less important. On the execution side, introducing sequential composition in the computations implies that there is no global time: two concurrent processes may independently perform computations of arbitrarily different lengths—i.e., we allow computations of the form $(u_1; \ldots; u_n \| v_1; \ldots; v_m)$. On the operation side, we enforce asynchrony—at least for objects of $\mathbf{U}_\Lambda(I, \Xi)$ built without abstraction—since the operation of processes on such "concrete" objects is sequentialized. Even the abstract objects of $\mathbf{U}_\Lambda(A, \Xi)$ do not strongly restrain the asynchrony of execution; for instance, a process using the communication structure port of CCS may perform a computation like

$$(\textit{send}; \ldots; \textit{send} \| \textit{receive} \| \ldots \| \textit{receive})$$

where each *receive* can wait for a corresponding *send.*

The meanings of *synchronization* in MEIJE/SCCS (cf. [2, 63]) and in our calculus are very different, partly because we do not have a uniform duration of computations in $\mathbf{P}_\Lambda(A,\Xi)$. More importantly, in MEIJE/SCCS one directly prescribes synchronization at the execution level, and one is thus able to define derived control structures from the primitive ones, as shown by De Simone in [63]. In our calculus, we have a more classical understanding of the synchronization problems, as regulating the concurrent accesses to shared resources. It is still unclear whether or not this is too restrictive a standpoint. One may wonder whether the strong notion of synchronization of MEIJE/SCCS is really consistent with the intuitive idea of a *distributed system.* On the other hand, it may be that our model of objects has some weaknesses with respect to the notion of reactive, or more accurately of *interactive system.* One must remark that, due to the abstraction rule, the operation of atom calls is purely *local* to an object; but one could imagine more "active" objects, for which the operation of an atom may trigger the execution of atom calls intended for some other agents. We leave all these hazy matters for further investigation.

A more definite question is that of the expressive power of abstraction. Abstraction provides us with the possibility of defining various communication mechanisms in the same language. We have suggested that one should *specify* abstract objects, so as to be able to *prove* that a defined object is a *correct implementation* of a given

specification. This remains to be formally stated. Some specifications cannot be carried out: for instance, no object can accept an operation $a;b$ without reacting also to a. We could prove that it is impossible to define an object enforcing a synchronization such as $(a;b\|c)$—without also allowing $(a\|b\|c)$. Here again, it is not clear whether this is a real deficiency.

Another study we plan to undertake concerns objects. The object constructors, namely declaration, parallel composition, sum, fixpoint and abstraction are perfectly meaningful as data type constructors. For instance, they allow us to deal with records such as $(l_1::s_1\| \dots \|l_k::s_k)$. But we do not have a syntax for primitive objects. Since they are states of transition systems, we could have allowed primitive instructions and sequential composition to build objects. But this is a rather ad hoc solution, which would suggest a "historical" view of objects: the state of an object would then represent its future, made of all the operations it will accept. However, we would like to have a more classical notion of object, and a less arbitrary notion of primitive instruction. A suitable framework could be that of event structures since they provide models for both process calculi and data type constructions, cf. [68, 70]. However, as Winskel observed some time ago, there is a mismatch: a process is an event structure, giving rise to a whole domain of computations, while a functional program is an element of a domain—one can also remark that a process or an object can be interpreted either as a whole transition system, or as a state of a transition system (note: Berry, Huet and Lévy [3, 34, 37] have shown that functional programs also determine an ordered set of computations, where the "events" are occurrrences of redexes, but this has not been much exploited in denotational semantics). Searching for "a good syntax" for objects and processes could bring us to a better understanding of the relationship between the semantics of sequential programs and that of concurrent and communicating systems.

Appendix A. Syntax

Finite terms. (i) $\mathbb{1}$ is a term of $\mathbf{T}(A)$ and every atom $a \in A$ is a term of $\mathbf{T}(A)$;
(ii) if p and q are terms of $\mathbf{T}(A)$, then so are $(p;q)$, $(p\|q)$ and $(p+q)$.

Terms. (i) $\mathbb{1}$ and every atom $a \in A$ are terms of $\mathbf{T}^{\text{rec}}(A)$;
(ii) if p and q are terms of $\mathbf{T}(A)$ and $\mathbf{T}^{\text{rec}}(A \cup Y)$ respectively, then $(p;q)$ is a term of $\mathbf{T}^{\text{rec}}(A \cup Y)$;
(iii) if p and q are terms of $\mathbf{T}^{\text{rec}}(A \cup Y)$ and $\mathbf{T}^{\text{rec}}(A \cup Y')$ respectively, then $(p\|q)$ and $(p+q)$ are terms of $\mathbf{T}^{\text{rec}}(A \cup Y \cup Y')$;
(iv) an identifier $x \in X$ is a term of $\mathbf{T}^{\text{rec}}(A \cup \{x\})$;
(v) if x is an identifier and p is a term of $\mathbf{T}^{\text{rec}}(A \cup Y)$, then $\mu x.p$ is a term of $\mathbf{T}^{\text{rec}}(A \cup Y - \{x\})$.

Actions. (i) $\mathbb{1}$ is a term of $\mathbb{D}(A)$ and every atom $a \in A$ is a term of $\mathbb{D}(A)$;
(ii) if p and q are terms of $\mathbb{D}(A)$, then so are $(p;q)$ and $(p\|q)$.

Processes. (i) $\mathbb{1}$ is a term of $\mathbf{P}_\Lambda(A, \Xi)$;

(ii) every atom or identifier $y \in A \cup X$ is a term of $\mathbf{P}_\Lambda(A \cup \{y\}, \Xi)$;

(iii) if p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$ and q is a term of $\mathbf{P}_\Lambda(A \cup Y', \Xi)$, then $(p;q)$, $(p \| q)$ and $(p+q)$ are terms of $\mathbf{P}_\Lambda(A \cup Y \cup Y', \Xi)$;

(iv) if $x \in X$ is an identifier and p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$, then $\mu x.p$ is a term of $\mathbf{P}_\Lambda(A \cup Y - \{x\}, \Xi)$;

(v) if $l \in \Lambda$ is a name and p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$, then $(l.p)$ is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$;

(vi) if $l \in \Lambda$ is a name, p is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$ and s is a term of $\mathbf{U}_\Lambda(A, \Xi)$, then $p\langle l{:}s \rangle$ is a term of $\mathbf{P}_\Lambda(A \cup Y, \Xi)$.

Objects. (i) Every identifier $y \in X$ is a term of $\mathbf{U}_\Lambda(A \cup \{y\}, \Xi)$ and every constant $o \in O$ is a term of $\mathbf{U}_\Lambda(A, \Xi)$;

(ii) if s is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$ and r is a term of $\mathbf{U}_\Lambda(A \cup Y', \Xi)$, then $(s \| r)$ and $(s+r)$ are terms of $\mathbf{U}_\Lambda(A \cup Y \cup Y', \Xi)$;

(iii) if $x \in X$ is an identifier and s is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$, then $\mu x.s$ is a term of $\mathbf{U}_\Lambda(A \cup Y - \{x\}, \Xi)$;

(iv) if $l \in \Lambda$ is a name and s is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$, then $(l{::}s)$ is a term of $U_\Lambda(A \cup Y, \Xi)$;

(v) if $z_1, \ldots, z_k \in Z$ are distinct atom identifiers, $p_1, \ldots, p_k$ are terms of $\mathbf{P}_\Lambda(A, \Xi)$, and s is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$, then $\{\alpha z_1, \ldots, z_k.p_1, \ldots, p_k\}s$ is a term of $\mathbf{U}_\Lambda(A \cup Y, \Xi)$.

Computations. (i) $\mathbb{1}$ is a term of $\mathbf{C}_\Lambda(A)$;

(ii) every atom $a \in A$ is a term of $\mathbf{C}_\Lambda(A)$;

(iii) if u and v are terms of $\mathbf{C}_\Lambda(A)$, then $(u;v)$ and $(u \| v)$ are terms of $\mathbf{C}_\Lambda(A)$;

(iv) if $l \in \Lambda$ is a name and u is a term of $\mathbf{C}_\Lambda(A)$, then $(l.u)$ is a term of $\mathbf{C}_\Lambda(A)$.

Appendix B. Axioms and theories

See Table 1.

Appendix C. Operational semantics

E1: atom

$$a \in A \vdash a \xrightarrow{a} \mathbb{1},$$

E2.1: *sequential composition* 1

$$p \xrightarrow{u} p' \vdash (p;q) \xrightarrow{u} (p';q),$$

Table 1

A0	$(p;(q;r))=((p;q);r)$	Υ					
U0	$(p;\mathbb{1})=p=(\mathbb{1};p)$		Γ				
A1	$(p\|(q\|r))=((p\|q)\|r)$	Υ		Δ			
U1	$(p\|\mathbb{1})=p=(\mathbb{1}\|p)$					Θ	
C1	$(p\|q)=(q\|p)$	Υ					
A2	$(p+(q+r))=((p+q)+r)$	Υ	Γ				
U2	$(p+\mathbb{1})=p=(\mathbb{1}+p)$				Ψ		
C2	$(p+q)=(1+p)$	Υ					
I	$(p+p)=p$						
B1	$a\bullet\mathbb{1}=a$ for $a\in A$						
B2	$\left(\sum_i \gamma_i\bullet p_i\right);q=\sum_i((\gamma_i\bullet p_i);q)$						Φ
B3	$(\gamma\bullet\mathbb{1});\left(\sum_j \beta_j\bullet q_j\right)=\gamma\bullet\left(\sum_j \beta_j\bullet q_j\right)+\sum_j(\gamma;\beta_j)\bullet q_j$						
B4	$\left(\gamma\bullet\left(\sum_i \gamma_i\bullet p_i\right)\right);q=\gamma\bullet\left(\sum_i(\gamma_i\bullet p_i);q\right)$						
B5	$\left(\sum_i \gamma_i\bullet p_i\|\sum_j \beta_j\bullet q_j\right)=\sum_i \gamma_i\bullet\left(p_i\|\sum_j \beta_j\bullet q_j\right)+\sum_{i,j}((\gamma_i\|\beta_j)\bullet(p_i\|q_j))+\sum_j \beta_j\bullet\left(\sum_u \gamma_i\bullet p_i\|q_j\right)$						
U3	$\mu x.\mathbb{1}=\mathbb{1}$						
U4	$(l.\mathbb{1})=\mathbb{1}$						
U5	$\mathbb{1}\langle l:s\rangle=\mathbb{1}$					Θ'	
D1	$(l.(p;q))=((l.p);(l.q))$						
D2	$(l.(p\|q))=((l.p)\|(l.q))$						

E2.2.1: *sequential composition* 2

$$p\equiv\mathbb{1},\ q\xrightarrow{v}q'\Vdash(p;q)\xrightarrow{v}q',$$

E2.2.2: *sequential composition* 3

$$p\xrightarrow{u}p'\equiv\mathbb{1},\ q\xrightarrow{v}q'\vdash(p;q)\xrightarrow{(u;v)}q',$$

E3.1: *parallel composition* 1

$$p \xrightarrow{u} p' \vdash (p \| q) \xrightarrow{u} (p' \| q),$$

E3.2: *parallel composition* 2

$$p \xrightarrow{u} p',\ q \xrightarrow{v} q' \vdash (p \| q) \xrightarrow{(u \| v)} (p' \| q'),$$

E3.3: *parallel composition* 3

$$q \xrightarrow{v} q' \vdash (p \| q) \xrightarrow{v} (p \| q'),$$

E4.1: *sum* 1

$$p \xrightarrow{u} p' \vdash (p + q) \xrightarrow{u} p',$$

E4.2: *sum* 2

$$q \xrightarrow{v} q' \vdash (p + q) \xrightarrow{v} q',$$

E5: *fixpoint*

$$p[\mu x.p/x] \xrightarrow{u} p' \vdash \mu x.p \xrightarrow{u} p',$$

E6: *qualification*

$$p \xrightarrow{u} p' \vdash (l.p) \xrightarrow{(l.u)} (l.p'),$$

E7: *block*

$$p \xrightarrow{u} p' \,\&\, u \copyright l,\ s \xmapsto{\lceil u/l \rceil} s' \vdash p\langle l{:}s\rangle \xmapsto{u \backslash l} p'\langle l{:}s'\rangle;$$

O0: *reaction*

$$(o, u, o') \in \xi \vdash o \xmapsto{\lceil u \rceil} o',$$

O1: *identity*

$$p \equiv \mathbb{1} \vdash s \xmapsto{\lceil p \rceil} s,$$

O2: *operation*

$$p \xrightarrow{u} p',\ s \xmapsto{\lceil u \rceil} s'' \xmapsto{\lceil p' \rceil} s' \vdash s \xmapsto{\lceil p \rceil} s',$$

O3.1: *parallel composition* 1

$$s \xmapsto{\lceil u \rceil} s' \vdash (s \| r) \xmapsto{\lceil u \rceil} (s' \| r),$$

O3.2: *parallel composition* 2

$$r \xmapsto{[v]} r' \vdash (s \| r) \xmapsto{[v]} (s \| r'),$$

O4.1: *sum* 1

$$s \xmapsto{[u]} s' \;\&\; u \not\equiv \mathbb{1} \vdash (s + r) \xmapsto{[u]} s',$$

O4.2: *sum* 2

$$r \xmapsto{[v]} r' \;\&\; v \not\equiv \mathbb{1} \vdash (s + r) \xmapsto{[v]} r',$$

O5: *fixpoint*

$$s[\mu x.s/x] \xmapsto{[u]} s' \vdash \mu x.s \xmapsto{[u]} s',$$

O6: *declaration*

$$s \xmapsto{[u]} s' \vdash (l::s) \xmapsto{[l.u]} (l::s'),$$

O7: *abstraction*

$$s \xmapsto{[u[\bar{p}/\bar{z}]]} s',\; u \in \mathbf{D}(\{\bar{z}\}) \vdash \{\alpha\bar{z}.\bar{p}\}s \xmapsto{[u]} \{\alpha\bar{z}.\bar{p}\}s'.$$

References

[1] L. Aceto, R. De Nicola and A. Fantechi, Testing equivalence for event structures, Tech. Rept. B4-6? Istituto di Elaborazione dell'Informa ione, CNR, Pisa (1986).

[2] D. Austry and G. Boudol, Algèbre de processus et synchronisations, *Theoret. Comput. Sci.* **30** (1984) 91-131.

[3] G. Berry and J.-J. Lévy, Minimal and optimal computations of recursive programs, *J. ACM* **26** (1979) 148-175.

[4] G. Berry and L. Cosserat, The ESTEREL synchronous programming language and its mathematical semantics, in: *Proc. Seminar on Concurrency*, Lecture Notes in Computer Science **197** (Springer, Berlin, 1984) 389-448.

[5] G. Boudol, Notes on algebraic calculi of processes, in: K. Apt, ed., *Logics and Models of Concurrent Systems*, NATO ASI Series F13 (1985) 261-303.

[6] G. Boudol, G. Roucairol and R. De Simone, Petri nets and algebraic calculi of processes, in: *Proc. Advances in Petri Nets 1985*, Lecture Notes in Computer Science **222** (Springer, Berlin, 1986) 41-58.

[7] G. Boudol, Communication is an abstraction, in: *Proc. Actes du Second Colloque* C^3 (1987) 45-63, and INRIA Res. Rept. 636.

[8] S. Brookes and W.C. Rounds, Behavioural equivalence relations induced by programming logics, in: *Proc. ICALP'83*, Lecture Notes in Computer Science **154** (Springer, Berlin, 1983) 97-108.

[9] S. Brookes, C.A.R. Hoare and A. Roscoe, A theory of communicating sequential processes, *J. ACM* **31** (1984) 560-599.

[10] R.H. Campbell and P. Jalotte, Atomic actions in concurrent systems, in: *Proc. 5th Internat. Conf. on Distributed Computing Systems* (1985) 184-191.

[11] P. Cartier and D. Foata, *Problèmes Combinatoires de Commutations et Réarrangements*, Lecture Notes in Mathematics **85** (Springer, Berlin, 1969).

[12] I. Castellani, P. Franceschi and U. Montanari, Labelled event structures: a model for observable concurrency, in: D. Bjørner, ed., *Formal Description of Programming Concepts II* (North-Holland, Amsterdam, 1983) 383-400.

[13] I. Castellani and M. Hennessy, Distributed bisimulations, Comput. Sci. Rept. 5-87, University of Sussex (1987).
[14] I. Castellani, Bisimulations and abstraction homomorphisms, *J. Comput. System Sci.* **34** (1987) 210-235.
[15] I. Castellani, Bisimulations for concurrency, Ph.D. Thesis, University of Edinburgh (1987).
[16] Ph. Darondeau and L. Kott, On the observational semantics of fair parallelism, in: *Proc. ICALP '83*, Lecture Notes in Computer Science **154** (Springer, Berlin, 1983) 147-159.
[17] P. Degano and U. Montanari, Distributed systems, partial orderings of events and event structures, in: M. Broy, ed., *Control Flow and Data Flow: Concepts of Distributed Programming*, NATO ASI Series F14 (1985) 7-106.
[18] P. Degano and U. Montanari, Specification languages for distributed systems, in: *Proc. 1st TAPSOFT*, Lecture Notes in Computer Science **185** (Springer, Berlin, 1985) 29-51.
[19] P. Degano, R. De Nicola and U. Montanari, *Partial ordering derivations for CCS*, in: *Proc. FCT 85*, Lecture Notes in Computer Science **199** (Springer, Berlin, 1985) 520-533.
[20] R. De Nicola, Extensional equivalences for transition systems, *Acta Inform.* **24** (1987) 211-237.
[21] E.W. Dijkstra, Cooperating sequential processes, in: F. Genuys, ed., *Programming Languages* (1968) 43-112.
[22] R.J. Duffin, Topology of series-parallel networks, *J. Math. Anal. Appl.* **10** (1965) 303-318.
[23] H.J. Genrich and E. Stankiewicz-Wiechno, A dictionary of some basic notions of net theory, in: W. Brauer, ed., *Net Theory and Applications*, Lecture Notes in Computer Science **84** (Springer, Berlin, 1980) 519-531.
[24] J.L. Gischer, Partial orders and the axiomatic theory of shuffle, Ph.D. Thesis, Stanford University (1984).
[25] R. van Glabbeek and F. Vaandrager, Petri net models for algebraic theories of concurrency, in: *Proc. PARLE Conf.*, Eindhoven, Lecture Notes in Computer Science **259** (Springer, Berlin, 1987) 224-242.
[26] U. Goltz and W. Reisig, The non-sequential behaviour of Petri nets, *Inform. and Control* **57** (1983) 125-147.
[27] J. Grabowski, On partial languages, *Fund. Inform.* **IV** (1981) 427-498.
[28] P.A. Grillet, Maximal chains and antichains, *Fund. Math.* **65** (1969) 157-167.
[29] M. Habib and R. Jegou, *N*-Free posets as generalizations of series-parallel posets, *Discrete Appl. Math.* **12** (1985) 279-291.
[30] M. Hennessy and R. Milner, Algebraic laws for nondeterminism and concurrency, *J. ACM* **32** (1985) 137-161
[31] M. Hennessy and R. De Nicola, Testing equivalences for processes, *Theoret. Comput. Sci.* **34** (1984) 83-133.
[32] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science (Prentice Hall, Englewood Cliffs, NJ, 1985).
[33] R.R. Hoogerwoord, An implementation of mutual inclusion, *Inform. Process. Lett.* **23** (1986) 77-80.
[34] G. Huet and J.-J. Lévy, Call-by-need computations in non-ambiguous linear term rewriting systems, IRIA-LABORIA Rept. 359 (1979).
[35] E.L. Lawler, Sequencing jobs to minimize total weighted completion time subject to precedence constraints, *Ann. Discrete Math.* **2** (1978) 75-90.
[36] E.L. Lawler, R.E. Tarjan and J. Valdes, The recognition of series parallel digraphs, *SIAM J. Comput.* **11** (1982) 298-313.
[37] J.-J. Lévy, Optimal reductions in the lambda calculus, in: J.P. Seldin and J.R. Hindley, eds., *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* (Academic Press, New York, 1980) 159-191.
[38] B. Liskov and R. Scheifler, Guardians and actions: linguistic support for robust, distributed programs, *ACM TOPLAS* **5** (1983) 381-404.
[39] D.B. Lomet, Process structuring, synchronization, and recovery using atomic actions, *SIGPLAN Notices* **12** (1977) 128-137.
[40] A. Mazurkiewicz, Concurrent program schema and their interpretations, in: *Proc. Aarhus Workshop on Verification of Parallel Programs*, Daimi PB-78, Aarhus University (1977).
[41] A. Mazurkiewicz, Traces, histories, graphs: instances of a process monoid, in: *Proc. MFCS'84*, Lecture Notes in Computer Science **176** (Springer, Berlin, 1984) 115-133.

[42] R. Milner, Program semantics and mechanized proofs, *Math. Centre Tracts* **82** (1976) 3-44.
[43] R. Milner, Synthesis of communicating behaviour, in: *Proc. MFCS'79*, Lecture Notes in Computer Science **64** (Springer, Berlin, 1979) 71-83.
[44] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science **92** (Springer, Berlin, 1980).
[45] R. Milner, On relating synchrony and asynchrony, CSR-75-80, Computer Science Dept., Edinburgh University (1980).
[46] R. Milner, Calculi for synchrony and asynchrony, *Theoret. Comput. Sci.* **25** (1983) 267-310.
[47] R. Milner, Lectures on a calculus for communicating systems, in: *Proc. Seminar on Concurrency*, Lecture Notes in Computer Science **197** (Springer, Berlin, 1985) 197-220.
[48] R. Milner, Process constructors and interpretations, in: *Proc. IFIP 86* (1986) 507-514.
[49] M. Nielsen, G. Plotkin and G. Winskel, Petri nets, event structures and domains, *Theoret. Comput. Sci.* **13** (1981) 85-108.
[50] E.-R. Olderog, Operational Petri net semantics for CCSP in: *Proc. Advances in Petri Nets '87*, Lecture Notes in Computer Science **266** (Springer, Berlin, 1987) 196-233.
[51] D. Park, Concurrency and automata on infinite sequences, in: *Proc. 5th GI Conf.*, Lecture Notes in Computer Science **104** (Springer, Berlin, 1981) 167-183.
[52] C.A. Petri, Non-sequential processes, GMD-ISF Rept. 77-05 (1977).
[53] G. Plotkin, A structural approach to operational semantics, Daimi FN-19, Aarhus University (1981).
[54] G. Plotkin, An operational semantics for CSP, in: D. Bjørner, ed., *Formal Description of Programming Concepts 2* (North-Holland, Amsterdam, 1983) 199-225.
[55] H. Plünnecke, *K*-density, *N*-density and finiteness properties, in: *Proc. Advances in Petri Nets '84*, Lecture Notes in Computer Science **188** (Springer, Berlin, 1984) 392-412.
[56] A. Pnueli, Linear and branching structures in the semantics and logics of reactive systems, in: *Proc. ICALP '85*, Lecture Notes in Computer Science **194** (Springer, Berlin, 1985) 15-32.
[57] V.R. Pratt, On the composition of processes, in: *Proc. 9th POPL* (1982) 213-223.
[58] V.R. Pratt, Modelling concurrency with partial orders, *Internat. J. Parallel Programming* **15** (1986) 33-71.
[59] W. Reisig, On the semantics of Petri nets, in: G. Chroust and E.J. Neuhold, eds., *Formal Models in Programming* (North-Holland, Amsterdam, 1985) 347-372.
[60] J. Riordan and C.E. Shannon, The number of two-terminal series-parallel networks, *J. Math. Phys.* **21** (1942) 83-93.
[61] C.E. Shannon, A symbolic analysis of relay and switching circuits, *Trans. Amer. Inst. Electr. Engrs.* **57** (1938) 713-723.
[62] M.W. Shields, Concurrent machines, *Comput. J.* **28** (1985) 449-465.
[63] R. de Simone, Higher level synchronising devices in MEIJE-SCCS, *Theoret. Comput. Sci.* **37** (1985) 245-268.
[64] D. Taubner and W. Vogler, The step failure semantics, in: *Proc. STACS '87*, Lecture Notes in Computer Science **247** (Springer, Berlin, 1987) 348-359.
[65] G. Viennot, Heaps of pieces: basic definitions and combinatorial lemmas, in: *Actes du Colloque de Combinatorie Enumérative*, Montreal (1985).
[66] J. Winkowski, Algebras of partial sequences, in: *Proc. FCT '77*, Lecture Notes in Computer Science **56** (Springer, Berlin, 1977) 187-196.
[67] J. Winkowski, Behaviours of concurrent systems, *Theoret. Comput. Sci.* **12** (1980) 39-60.
[68] G. Winskel, Events in computation, Ph.D. Thesis, Edinburgh University (1980).
[69] G. Winskel, Event structure semantics for CCS and related languages, *9th ICALP*, Lecture Notes in Computer Science **140** (Springer, Berlin, 1982) 561-576.
[70] G. Winskel, Event structures, in: *Proc. Advances in Petri Nets '86*, Lecture Notes in Computer Science **255** (Springer, Berlin, 1987) 325-392.