What is a State of a System? (an outline)

Uri Abraham
Department of mathematics and computer science
Ben-Gurion University, Be'er Sheva, ISRAEL.

Abstract

We try to answer this question in defining a state as a kind of a pomset. The notion of a pomset is extended by adding to the unary predicates unary functions of bounded depth. We show the equivalence of two approaches to modeling concurrency: the sequential point of view (which is based on states) and the partial-order point of view (based on events). This note points to a possible approach for defining the semantics of communicating processes. We define and show how to work with a natural subclass of information systems, which were introduced by Scott to give denotational domains.

1 Preface

For sequential processes, a state is simply the complete information on all the variables and control positions at a particular instant. For concurrent communicating processes, the notion *state* has no such simple definition, unless severe limitations on concurrency are imposed. Considering its central place, it is surprising that the general notion *state* has received so little attention.

In this note the notion 'pomset' is extended in such a way that a state is just a pomset with no temporal precedence relation on its events—it is the description of an instant (Section 3.3). This allows to demonstrate within a single framework the duality between two approaches to modeling concurrency: the one based on events and the other on states. For one direction in this duality, any pomset is described as a serial concatenation of its states (Section 4). And for the other direction, a sequence of abstractly given states give rise to a partial order of events (Section 2).

This approach to the notion of a state, and the construction of domains by Scott [1982], are used here to give a (denotational) semantics to a very simple language for concurrent processes. In fact, we modify to some extent the original definition of information systems of Scott, and provide (in Section 6) an account of a simplified version that suffices for our use.

2 From States to Events

The prime notion in the approach to modeling based on pomsets is the 'event', with its characteristics and temporal position. In this section we show how pomsets arise naturally, even if the starting point is the abstract notion of a state rather than that of an event.

Assuming global time, reality can be described as a sequence of states. A *run* is a structure $\mathcal{R} = \langle N, <, F \rangle$, where N (the set of integers or a finite initial segment of it) is the sequence of states, $<$ is the usual order on the integers and is intuitively seen here as the temporal precedence order on N. F is a finite collection of unary predicates on N. The meaning of $i < j$ is that state i appeared before state j. $P(i)$ means that P is true at state i. Since the predicates are unary, the second-order monadic theory of ω expresses all that we may wish to say about runs.

What is an event? Usually, an event is defined as the object that caused the transition from one state to its successor. Such an approach is suitable when concurrency is modeled by interleaving atomic actions and an elementary object is the triple $S_1 \xrightarrow{e} S_2$ which indicates that event e has caused the change from state S_1 to state S_2. We take here an alternative approach: An event occurrence is a constant and continuous thread through the succession of changing states. A formal definition follows. Some predicates in F denote the occurrences of events. Given such a predicate P, $P(i)$ indicates that the event denoted by P occurs at the i-th state. It is assumed that the description given by P is so complete that if $P(i)$ holds, then only one event of type e_P (the type of P) occurs at the i-th state. The point is that if two events with property P occurred at i, then they are indistinguishable and hence are the same event. Now suppose both $P(i)$ and $P(i+1)$ are true, how do we know then whether it is the same occurrence of the event that appeared in the two successive states? Could it not be that two distinct occurrences of events of type P happened at state i and $i+1$? It is necessary therefore to be able to tell whether an event is new or is a continuation ot its occurrence in the previous state. There are several ways to do that, and later we will suggest one way. Let us only assume now that somehow the predicates in F can tell whether $P(i)$ and $P(i+1)$ represent the same or two distinct occurrences.

This leads to the following. Let $\mathcal{R}$ be a run. For each $P \in F$ that is supposed to describe occurrences of events of type e_P, let us define "an occurrence of e_P" to be a maximal interval I in N such that (1) $P(i)$ holds for all $i \in I$, and (2) for $i, i+1 \in I$, $P(i)$ and $P(i+1)$ represent the same occurrence.

Let E be the collection of all event intervals thus obtained as occurrences of e_P for some $P \in F$. A partial order $\prec$ is defined on $I, J \in E$ by:

$$I \prec J \ \text{ iff } \ \forall x \in I \ \forall y \in J (x < y).$$

The set of event occurrences of a fixed type e_P are linearly ordered by $\prec$. That is, E is the union of finitely many chains (since F is finite).

The predicates of F can be defined on E as well: For $I \in E$ of type e_P and for $Q \in F$ that has the form "property R holds for the event of type e_P", $R(I)$ holds iff $\forall x \in I,\ Q(x)$. The structure $(E, \prec, F)$ thus obtained will be defined in the following section as a pomset (Partially Ordered MultiSet).

3 On Pomsets

The pomsets defined in this paper combine features from the original definitions of Gischer [1989] and Pratt [1986], and from Lamport's system executions [1986], with the addition of some other features (mainly the addition of functions). First, we present a restricted version of pomsets that have only predicates, and in the following section functions are added.

A partial order $<_{\mathcal{S}}$ on a set $E_{\mathcal{S}}$ is called an *interval order* if the following (Russell-Wiener property) holds:

$$a <_{\mathcal{S}} b,\ b \not>_{\mathcal{S}} c, \text{ and } c <_{\mathcal{S}} d \text{ imply } a <_{\mathcal{S}} d.$$

By Wiener [1914], any interval order can be represented by intervals of a linear order. An outline of the proof follows (see Fishburn [1986], or Anger [1989] or Abraham, Ben David and Magidor [1990]).

The intuition of Russell and Wiener is that instants (points of time) can be retrieved from the events (which are extended in time). Recall that an antichain in a partially ordered set, $(E_{\mathcal{S}}, <_{\mathcal{S}})$, is a non empty collection of pairwise incomparable members. We are assuming that $E_{\mathcal{S}}$ is countable, and each $e \in E_{\mathcal{S}}$ has finitely many predecessors, and every antichain is finite.

Wiener's representation result begins by collecting all maximal antichains of $(E_{\mathcal{S}}, <_{\mathcal{S}})$ into a set M, the set of 'instants'. In our case, since we assume that every antichain is finite, M is countable. Then define for $m_1, m_2 \in M$,

$$m_1 <_M m_2 \text{ iff } \exists a \in m_1,\ \exists b \in m_2\ (a <_{\mathcal{S}} b).$$

Assuming the Russell-Wiener property, the relation $<_M$ is a linear order of M, and if M is infinite then its order-type is ω (by the finiteness of predecessors assumption).

The representation is the function μ defined on $E_{\mathcal{S}}$ by:

$$\mu(e) = \{m \mid e \in m\}.$$

$\mu(e)$ is a convex interval in $<_M$, and

$$e_1 <_{\mathcal{S}} e_2 \text{ iff } \mu(e_1) \prec \mu(e_2).$$

Where $\prec$ is as the order defined in Section 2, but now on the intervals of M. We claim that if $m_1 <_M m_2$ are maximal antichains, and m_2 is the successor of m_1 in $<_M$, then

$$\forall x \in m_1 - m_2 \ \forall y \in m_2 - m_1 \ (x <_{\mathcal{S}} y).$$

Indeed, if x and y as above were incomparable in $<_S$, then $m_1 \cap m_2 \cup \{x, y\}$ could be extended to a maximal antichain m, and necessarily $m_1 <_M m <_M m_2$.

Definition 3.1 A *pomset* (with unary predicates) is a structure $\mathcal{P} = (E_{\mathcal{P}}, <_{\mathcal{P}}, F_{\mathcal{P}})$ where

(1) $(E_{\mathcal{P}}, <_{\mathcal{P}})$ is an interval order. (We will use the convention that sets $a \leq_{\mathcal{P}} b$ iff $a <_{\mathcal{P}} b$ or $a = b$.)

(2) For every $e \in E_{\mathcal{P}}$, $\{ x \in E_{\mathcal{P}} \mid x \not>_{\mathcal{P}} e\}$ is finite. (This is Lamport's finiteness condition).

(3) $F_{\mathcal{P}}$ is a finite collection of unary predicates on $E_{\mathcal{P}}$.

(4) There is a finite bound k on the size of antichains in $(E_{\mathcal{P}}, <_{\mathcal{P}})$. In fact, there is in $F_{\mathcal{P}}$ a decomposition of $E_{\mathcal{P}}$ into k many chains (linear subsets).

Let us emphasize that, primarily, we view a pomset as a description of reality, one world out of many possible, and not as a prescription of how things should be. Thus $a <_{\mathcal{P}} b$ means that a happens before b, and not that a caused b or that b is bound to follow a.

Sometimes it is convenient to include in $F_{\mathcal{P}}$ unary functions as well, from $E_{\mathcal{P}}$ into some fixed (finite) range. Since such functions can be represented with finitely many unary predicates, this is just a stylistic addition.

Condition (4) is natural when one puts a finite limit on all possible resources; for example if there are finitely many serial processors working concurrently.

In the following section and in Section 3.3 we will extend this notion of a pomset in two directions: by adding functions from $E_{\mathcal{P}}$ to $E_{\mathcal{P}}$, and by adding initial and final events. There is clearly also the need to represent in $E_{\mathcal{P}}$ those fixed object for which it makes no sense to apply the precedence relation. For example, the program statements and control positions, or the possible values of the variables. However, for notational simplicity, we will not do this here.

3.1 Pomsets with functions

Experience with protocols and correctness proofs has taught us the advantage of supplementing pomsets with unary functions from $E_{\mathcal{S}}$ to $E_{\mathcal{S}}$. However, in order not to lose the decidability of the theory of pomsets given by Büchi's theorem, only a limited family of functions should be allowed: non-increasing functions with bounded depth (defined below). It turns out that only definable functions are thus added, and the functionless pomsets suffice—at least in theory. In practice, we find that the addition of functions can help in understanding systems. As an example, see the definition of a queue given in Section 3.4.

For simplicity, let us restrict the following discussion to the monadic theory of the order of N. Let $\mathcal{R} = \langle N, <, F\rangle$ be a monadic structure. (i.e., $<$ is the usual order relation on the set of integers, N, and F is a collection of unary predicates and unary functions with finite range.) A function $f : A \to N$, where $A \subseteq N$, is *definable* in $\mathcal{R}$ iff there is a 2nd order monadic formula $\varphi(x, y)$ such that $y = f(x)$ iff $\mathcal{R} \models \varphi(x, y)$.

A *decreasing* function is a function satisfying $f(x) < x$ for all $x \in dom(f)$. Some decreasing functions have the following simple definition. A subset, $B \subseteq N$, is given, and for every $x \in A$:
$f(x) = y$ *iff* $y \in B$ *is the rightmost member of* B *such that* $y < x$.

A slightly more general definition of a function allows for several such B's. Let us define a decreasing function f to be *simply definable* in $\mathcal{R}$ if $f : A \to N$, and there are, in F, $B_1, \ldots, B_n \subseteq N$, and $g : A \to \{1, \ldots, n\}$ such that:
$f(x) = y$ iff $y \in B_{g(x)}$ *is the rightmost member* $y' \in B_{g(x)}$ *such that* $y' < x$.

Let us say that $f : A \to N$, $A \subseteq N$, is a function of *depth* k, if for every $e \in N$ there is a set $T \subseteq [0, \ldots, e]$ of cardinality $\leq k$ such that:

$$\forall x > e\ (f(x) \leq e \Rightarrow f(x) \in T).$$

If for some k, f has depth k, then f is said to be "of bounded depth".

The following was confirmed in conversation with M. Magidor.

For a decreasing function $f : A \to N$, (1) implies (2) and (2) implies (3).

(1) f is definable in $\mathcal{R}$.

(2) f is of bounded depth.

(3) f is simply definable in some extension of $\mathcal{R}$ (obtained by adding unary predicates).

We now generalize the discussion from monadic structures to the setting of a pomset $\mathcal{S} = (E_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}})$. Let us say that a function $f : A \to E_{\mathcal{S}}$, $A \subseteq E_{\mathcal{S}}$, is non-increasing if, for every e, $f(e) \not>_{\mathcal{S}} e$. Say f has depth k, if for every

$e \in E_{\mathcal{S}}$ there is a set T of size $\leq k$, such that if $e \not\geq_{\mathcal{S}} x$ and $f(x) \leq_{\mathcal{S}} e$, then $f(x) \in T$. In this case, since there is a bound, b, on the size of the antichains, for every instant m (a maximal antichain) there is a set U of size $\leq k \times b$ such that: If $x \geq e$ for some $e \in m$, and $f(x) < e'$ for some $e' \in m$, then $f(x) \in U$.

Definition 3.2 A pomset (with functions) is a pomset as in Definition 3.1, but now with (finitely many) non increasing functions of bounded depth. We will call such a pomset *plain* to distinguish it from the full pomsets defined in Section 3.3 (by adding initial and final events).

3.2 Initial and final segments of pomsets (intuitive remarks)

An initial segment of a pomset $\mathcal{S}$ (with functions) represents the run up to a certain instant. A final segment is the run from some instant on. Both are subpomsets. In the following we are going to motivate the distinction between *finished* and *unfinished* events and the introduction of initial events.

Given a pomset S, and an instant m (a maximal antichain) let us form

$$\leq (m) = \{\ x \in E_{\mathcal{S}} \mid x \leq_{\mathcal{S}} a \text{ for some } a \in m\},$$

$$\geq (m) = \{\ y \in E_{\mathcal{S}} \mid a \leq_{\mathcal{S}} y \text{ for some } a \in m\}.$$

Now let m_1 and m_2 be a maximal antichain and its successor in the order $<_M$ defined on the maximal antichains. Then the following hold:

1. $\forall a \in m_1 \setminus m_2\ \forall b \in m_2 \setminus m_1\ (a <_{\mathcal{S}} b)$
2. $\leq (m_1) \cup \geq (m_2) = E_{\mathcal{S}}$.

In fact, for the ongoing discussion it suffices to assume that m_1 and m_2 are antichains (not necessarily maximal) that satisfy these conditions.

We wish to form the initial pomset $\mathcal{S}_1$ with set of events $E_{\mathcal{S}_1} = \leq (m_1)$, and the final pomset $\mathcal{S}_2$ with set of events $E_{\mathcal{S}_2} = \geq (m_2)$. It is required of such a definition that the concatenation $\mathcal{S}_1; \mathcal{S}_2$ (to be defined later) gives back $\mathcal{S}$. For that, we need to be able to identify those events that are common to $\mathcal{S}_1$ and to $\mathcal{S}_2$, in order to unify them in the concatenation. It is easy to see that a is both in $\leq (m_1)$ and $\geq (m_2)$ iff $a \in m_1 \cap m_2$. The way we single out the events in $m_1 \cap m_2$ is by declaring them to be 'unfinished' in $\mathcal{S}_1$ (this is so because any $a \in m_1 \cap m_2$ in $\mathcal{S}_1$ continues to occur in $\mathcal{S}_2$).

So we assume a predicate in $F_{\mathcal{S}}$ that tells us about each event whether it is finished or not. An unfinished event in an initial segment may become finished in a longer segment. (For all other predicates in F we assume that the truth value of $P(x)$ does not change in the passage from a pomset to a subpomset —this is of course the definition of a substructure). The only role of the finished/unfinished distinction is to enable the identification of an unfinished event in a pomset with its continuation in an extension of that pomset. In an infinite pomset all events are finished, and in a finite pomset unfinished events are always maximal. So if $a <_{\mathcal{S}} b$ then a is finished.

Now let us look at the final segment $\geq (m_2)$ and try to generate the appropriate pomset by restricting the predicates and functions of $F_{\mathcal{S}}$ to $E_{\mathcal{S}_2}$. It may well be that $E_{\mathcal{S}_2}$ is not closed under the functions in $F_{\mathcal{S}}$. This shows the need for *initial events*. These are 'prehistoric' events, not present in $E_{\mathcal{S}_2}$ but used to explain its behavior. They will form the initial pomset of $\mathcal{S}_2$ which will also be the final pomset of $\mathcal{S}_1$. Typically, the final pomset includes all the relevant information needed for the system to resume its execution. In other words, the final pomset of $\mathcal{S}_1$ is the initial pomset of any possible continuation. Since the functions in $F_{\mathcal{S}}$ are of bounded depth, the number of possible initial states will be finite. Let us now formally develop these intuitive considerations.

3.3 Pomsets with initial and final events

A pomset with initial and final events is a structure $\mathcal{S} = (E_{\mathcal{S}}, I_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}}, T_{\mathcal{S}})$ consisting of

1. a set of events $E_{\mathcal{S}}$,
2. a (finite) set of *initial events* $I_{\mathcal{S}}$,
3. a precedence relation $<_{\mathcal{S}}$ (defined on $E_{\mathcal{S}} \cup I_{\mathcal{S}}$),
4. a finite collection, $F_{\mathcal{S}}$, of (partial) unary predicates and non-increasing functions of bounded depth, and
5. a finite set $T_{\mathcal{S}} \subseteq E_{\mathcal{S}} \cup I_{\mathcal{S}}$, of "final events".

Let us call the events in $E_{\mathcal{S}}$ 'actual' to distinguish them from the 'prehistoric' events in $I_{\mathcal{S}} \setminus E_{\mathcal{S}}$. It is possible that the set $I_{\mathcal{S}} \cap E_{\mathcal{S}}$ is nonempty, but this intersection is always an antichain in $<_{\mathcal{S}}$. If $a \in I_{\mathcal{S}} \setminus E_{\mathcal{S}}$ and $b \in E_{\mathcal{S}} \setminus I_{\mathcal{S}}$ then $a <_{\mathcal{S}} b$. $I_{\mathcal{S}}$ is an initial segment, and $E_{\mathcal{S}}$ a final segment in $<_{\mathcal{S}}$; that means that if $x <_{\mathcal{S}} y$ then $x \in E_{\mathcal{S}}$ implies $y \in E_{\mathcal{S}}$, and $y \in I_{\mathcal{S}}$ implies $x \in I_{\mathcal{S}}$.

The functions and relations are on $E_{\mathcal{S}} \cup I_{\mathcal{S}}$. In all of our applications, the functions are partial functions with the property that for any set X the union of the ranges $f''X$ for $f \in F_{\mathcal{S}}$ is closed (under all the functions). If $T_{\mathcal{S}}$ is undefined or $T_{\mathcal{S}} = \emptyset$ then it may be omited from $\mathcal{S}$. This is the case when $E_{\mathcal{S}}$ is infinite. The precedence relation is an interval order, and in fact all the requirements of Definition 3.1 are made here too. It is possible to omit $I_{\mathcal{S}}$ and $T_{\mathcal{S}}$ from the definition, and in this case $\mathcal{S}$ is called a *plain* pomset. For emphasize, a pomset with initial and final events may be called *full*.

There is a distinguished *finished/unfinished* predicate in $F_{\mathcal{S}}$ defined on the actual events. In an infinite pomset the events of $E_{\mathcal{S}}$ are all finished, but some events may be unfinished in a finite pomset. If $a <_{\mathcal{S}} b$ then a is finished (so the unfinished events form an antichain). When the predicate $T_{\mathcal{S}}$ is present (in a full pomset), the unfinished events of $E_{\mathcal{S}}$ are all in $T_{\mathcal{S}}$, but it is possible for $T_{\mathcal{S}}$ to also have finished events. (The events in $I_{\mathcal{S}} \setminus E_{\mathcal{S}}$ are considered to be finished.)

Definition 3.3 A pomset $\mathcal{S}$ in which $E_{\mathcal{S}}$ is an antichain (possibly empty) is called a *state* (a *pomset state*, when other notions of states are around). If $E_{\mathcal{S}} = \emptyset$, then $\mathcal{S}$ is called an 'eventless' state (or pomset).

For each pomset with initial/final events, two plain pomsets are defined: its initial, and final pomsets. Let us first define the initial pomset of $\mathcal{S}$.

Given a pomset $\mathcal{S} = (E_{\mathcal{S}}, I_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}}, T_{\mathcal{S}})$, the *initial pomset* of $\mathcal{S}$ is the plain pomset $\mathcal{U} = (I_{\mathcal{S}}, <_{\mathcal{S}} \lceil I_{\mathcal{S}}, F_{\mathcal{S}})$, obtained by taking $E_{\mathcal{U}} = I_{\mathcal{S}}$ with $<_{\mathcal{U}}$ and $F_{\mathcal{U}}$ being the restrictions of $<_{\mathcal{S}}$ and $F_{\mathcal{S}}$ to $I_{\mathcal{S}}$; except that the unfinished events of $\mathcal{U}$ are exactly $I_{\mathcal{S}} \cap E_{\mathcal{S}}$. (Intuitively this is so since these events continue to be in $E_{\mathcal{S}}$).

The *final pomset* of $\mathcal{S}$ is naturally defined by restricting $<_{\mathcal{S}}$ and $F_{\mathcal{S}}$ to $T_{\mathcal{S}}$: That is $\mathcal{F} = (T_{\mathcal{S}}, (<_{\mathcal{S}} \lceil T_{\mathcal{S}}), F_{\mathcal{S}} \lceil T_{\mathcal{S}})$. Thus the final pomset of $\mathcal{S}$ is the plain pomset $\mathcal{F}$ obtained by transforming the final events $T_{\mathcal{S}}$ into the events of $\mathcal{F}$. The intuition is that the final pomset of $\mathcal{S}$ constitutes the initial pomset of any continuation of $\mathcal{S}$. The unfinished events of $\mathcal{F}$ are the unfinished events of $\mathcal{S}$ (required to be in $T_{\mathcal{S}}$).

Let $\mathcal{S}$ be a pomset, and m_1 and m_2 be an instant and its successor in $\mathcal{S}$ (or just two antichains that satisfy 1 and 2 of Section 3.2). The initial segment, $\mathcal{S}_1$, generated by m_1 and the final segment, $\mathcal{S}_2$, generated by m_2 (that we intuitively defined in Section 3.2) can now be completely defined. The final events of $\mathcal{S}_1$ are exactly the initial events of $\mathcal{S}_2$: these include the unfinished events of $E_{\mathcal{S}_1}$ (that is, $m_1 \cap m_2$), and all the events of the form $f(x)$ for $f \in F_{\mathcal{S}}$, where $x \in \geq (m_2)$, but $f(x) \notin \geq (m_2)$. (It is assumed that the set thus obtained is closed under the functions of $F_{\mathcal{S}}$.)

The initial pomset of $\mathcal{S}_2$ equals the final pomset of $\mathcal{S}_1$, and the concatenation of $\mathcal{S}_1$ with $\mathcal{S}_2$ will be defined in Section 4 so as to give back $\mathcal{S}$.

3.4 An example: specifying a queue

As an example to the use of functions in pomsets, let us specify a bounded queue for one *Producer* and two *Consumers*, where an item in the queue withdrawn by one of the *Consumers* is no longer available to the other. Thus if two dequeue operations are executed, by $Consumer_1$ and $Consumer_2$, it is never the same item that is delivered to them (it may have the same value of course). The queue is supposed to have a bounded capacity, k. The operations of the *Producer* are called enqueue, and those of the *Consumers* are called dequeue events (thus, since we are only interested in the queue specifications, the producing and consuming events are disregarded).

The specification (or description) of the queue is given by the collection of all pomsets that behave correctly. (A collection of pomsets is called a 'system'.) This queue specifying system is determined by the following list of 'axioms' concerning a pomset $\mathcal{S} = (E_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}})$.

We will be using the relation $a <_{\mathcal{S}} end(b)$ on events in $E_{\mathcal{S}}$. This means that

$$\forall x \in E_{\mathcal{S}} \ (b <_{\mathcal{S}} x \Rightarrow a <_{\mathcal{S}} x).$$

Similarly, $begin(a) <_{\mathcal{S}} b$ means that $\forall x <_{\mathcal{S}} a,\ x <_{\mathcal{S}} b$.
QUEUE SPECIFICATIONS (the parenthetical explanations are only for intuition):

1. Each of the processes in $\mathcal{S}$, *Producer*, $Consumer_1$, and $Consumer_2$ (subsets of $E_{\mathcal{S}}$), are serial (linearly ordered by $<_{\mathcal{S}}$). Put $Consumer = Consumer_1 \cup Consumer_2$. The events in *Producer* are called 'enqueue', and those in *Consumer*, 'dequeue'.

2. There are in $F_{\mathcal{S}}$ a predicate 'successful', and two functions: f and *Value*. (Every event is of finite duration (i.e., it is *terminating*) and it returns a value indicating its status, successful or not. Intuitively, a *Producer* enqueue is not successful when the queue has already k items, and a dequeue event is not successful when the queue is empty. $Value(e)$ is the data value of the enqueue or dequeue event e. The function f is defined on the successful dequeue events, and $f(c) = p$ means that c obtained the item deposited by the (successful) enqueue event p. There is more than one way of making this intuition precise, and we have chosen one way as an example.)

3. f is defined on the successful dequeue events, and is taking values in successful enqueue events.

   (a) f is one-to-one, and $f(a) <_{\mathcal{S}} a$ for all a.

   (b) f is order-preserving:
   $$a <_{\mathcal{S}} b \Rightarrow f(a) <_{\mathcal{S}} f(b).$$

   (c) $Value(a) = Value(f(a))$.

4. For every successful enqueue event p the following set has cardinality $< k$:
   $\{q \in Producer \mid q <_{\mathcal{S}} p,\ q\ \textit{is a successful enqueue, and there is no}\ c \in Consumer\ \textit{such that}\ \ q = f(c)\ \textit{and}\ c <_{\mathcal{S}} end(p)\}$.

5. For every *un*successful enqueue event p the following set has cardinality $\geq k$:
   $\{q \in \mathit{Producer} \mid q \textit{ is a successful enqueue, } q <_{\mathcal{S}} p, \textit{and there is no } c \in \mathit{Consumer},\ c <_{\mathcal{S}} p,\ \textit{such that } q = f(c)\}$.

6. For every successful *Consumer* dequeue c, if $p <_{\mathcal{S}} f(c)$ is a successful enqueue event, then $p = f(c')$ for some dequeue c'.

7. For every unsuccessful dequeue event c, there is, for every successful enqueue $p <_{\mathcal{S}} c$, a dequeue event a such that $f(a) = p$ and $\mathit{begin}(a) <_{\mathcal{S}} c$.

3.5 Environments

In the traditional view, a state is a map form the set of variables into the set of possible values. To not overload the word 'state', let us call such a map a 'variable environment'. This traditional view capture an intuition not present in our definition of a state as a pomset with no precedence relation on its events; an intuition which is particularly useful when dealing with the semantics of programming languages, because the execution of a program depends on the environment.

It therefore makes sense to incorporate the traditional view in our definition, and this can be done simply by adding to a pomset $\mathcal{S}$ two (partial) functions

$$I\_ENV,\ F\_ENV : \mathit{Var} \rightarrow \mathit{Value}$$

where *Var* is the set of variables, and *Value* is the set of possible values; *I_ENV* and *F_ENV* are the initial and final variable environments.

These functions are added to $F_{\mathcal{S}}$, but for emphasize we can write them externally, so that the resulting pomset is represented as

$$\mathcal{A} = (\mathcal{S}, I\_ENV, F\_ENV).$$

In an eventless pomset the initial and final environments must be the same since there are no events.

4 Operations on Pomsets: Concatenation and Concurrent unions

In this section we first define the concatenation of two pomsets and show how every pomset can be decomposed into a sequence of states. Then we define the concurrent union of two pomsets.

4.1 Concatenation

Let $\mathcal{A}$ and $\mathcal{B}$ be two pomsets (with $\mathcal{A}$ finite), and suppose that $\mathcal{S}$ is a plain pomset that occurs both as a final pomset of $\mathcal{A}$ and as an initial pomset of $\mathcal{B}$. Then $\mathcal{A};\mathcal{B}$, which is the concatenation of $\mathcal{A}$ followed by $\mathcal{B}$, can be defined by identifying these two occurrences of $\mathcal{S}$, and requiring that, except for this identification, all the events of $\mathcal{A}$ precede those of $\mathcal{B}$. The initial events of $\mathcal{A};\mathcal{B}$ are the initial events of $\mathcal{A}$, and its final events are those of $\mathcal{B}$.

To be more specific, $\mathcal{A};\mathcal{B} = \mathcal{C}$ is defined thus. Set $I_{\mathcal{C}} = I_{\mathcal{A}}$. To obtain $E_{\mathcal{C}}$, take first the disjoint union of $E_{\mathcal{A}}$ with $E_{\mathcal{B}}$, and then identify events in $E_{\mathcal{A}}$ and $E_{\mathcal{B}}$ as follows. Let i_1 and i_2 be the embeddings of $\mathcal{S}$ as a final pomset of $\mathcal{A}$, and an initial pomset of $\mathcal{B}$ respectively. So i_1 is taking $E_{\mathcal{S}}$ onto $T_{\mathcal{A}}$, and i_2 is onto $I_{\mathcal{B}}$. Now, if $s \in E_{\mathcal{S}}$ is such that $i_2(s) \in E_{\mathcal{B}}$, then $i_1(s)$ and $i_2(s)$ are identified. So the elements of $E_{\mathcal{B}}$ that are thus identified with members of $\mathcal{A}$ are exactly those in $I_{\mathcal{B}} \cap E_{\mathcal{B}}$, and these were defined to be the unfinished events of the initial pomset of $\mathcal{B}$. Hence it is exactly the unfinished events of $\mathcal{A}$ that are identified with actual elements of $\mathcal{B}$. (Recall that the unfinished events in the final pomset of $\mathcal{A}$, namely in $\mathcal{S}$, were defined to be the unfinished events of $\mathcal{A}$.)

The final events of $\mathcal{C}$ are defined to be $T_{\mathcal{B}}$, the final events of $\mathcal{B}$ (in case $T_{\mathcal{B}}$ contains some members of $I_{\mathcal{B}}$, these are identified, via $\mathcal{S}$, with the corresponding members of $I_{\mathcal{A}} \cup E_{\mathcal{A}}$). The precedence relation $<_{\mathcal{C}}$ is defined by first taking the union of $<_{\mathcal{A}}$ with $<_{\mathcal{B}}$ and then extending the relation so that for $a \in (E_{\mathcal{A}} \cup I_{\mathcal{A}}) \setminus E_{\mathcal{B}}$ and $b \in E_{\mathcal{B}} \setminus I_{\mathcal{B}}$, $a <_{\mathcal{C}} b$ holds. The Russell-Wiener property holds in $\mathcal{A};\mathcal{B}$.

It is not difficult to prove that concatenation is an associative operation on pomsets.

In particular, it is possible now to define the concatenation of a sequence $\langle \mathcal{S}_i | i < i_0 \rangle$ (i_0 is finite or ω) of states. Recall that a state is a pomset with $E_{\mathcal{S}_i}$ forming an antichain, and it has its own initial and final (plain) pomsets. It is assumed that for every i the final pomset of $\mathcal{S}_i$ equals the initial pomset of $\mathcal{S}_{i+1}$ in order for this concatenation sequence to be defined.

Now, any pomset can be represented as the concatenation of a sequence of its states. The coarser way to do that is to look at the sequence of instants. Each instant (maximal antichain) defines a state, and these instants are linearly ordered and their concatenation gives back the pomset. We are back to our starting point where a run was given as a sequence of states. It should be observed that there are finer ways to decompose a pomset into states. All these decompositions give back the pomset, and intuitively they correspond to decompositions obtained with clocks of varying ticking rates. Graphically, if the events of the pomset are represented by intervals, then each decomposition corresponds to a sequence of points where each point corresponds to an antichain (not necessarily maximal), and different points may correspond to the same antichain.

To take the simplest example, suppose the pomset $\mathcal{S}$ contains a single serial process, enumerated as $e_0, e_1 \ldots$, in increasing $<_{\mathcal{S}}$ order. The coarser decomposition of $\mathcal{S}$ is into the states $\mathcal{S}_i$ that contain the antichain e_i (a single event). Finer decompositions could be described graphically by first representing the events e_i's as intervals and then picking a sequence of points that intersect each interval in one or more points and may also contain points falling in between adjacent intervals.

Let us give another example to illustrate a nonsequential process. Suppose the pomset $\mathcal{S}$ consists of three events, $E_{\mathcal{S}} = \{a, b, c\}$, and the only temporal relation is $a <_{\mathcal{S}} c$. There are thus only two maximal antichins, $m_1 = \{a, b\}$, and $m_2 = \{c, b\}$. The coarser decomposition is therefore into two states, and then no state corresponds, for example, to the moment when a has finished, c has not yet begun and b is ongoing. An example of a finer decomposition into states is $\mathcal{S}_1; \ldots; \mathcal{S}_5$, where the actual events of the five states are: $\{b\}$, $\{a, b\}$, $\{b\}$, $\{c, b\}$, and $\{c\}$. In this concatenation, b begins before a, and ends before c ends.

4.2 Concurrent unions

We shall first define the term 'restriction of a pomset', and then define the concurrent union of two pomsets that have the same restrictions: it is the collection of pomsets obtained by 'running' the two pomsets concurrently while identifying the events in this common restriction.

Restrictions: Let $\mathcal{S} = (E_{\mathcal{S}}, I_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}})$ be a pomset. If $A \subseteq E_{\mathcal{S}} \cup I_{\mathcal{S}}$, and G is a set of relation and function symbols, then the *restriction* of $\mathcal{S}$ to A and G is the pomset obtained by restricting the functions and predicates of $F_{\mathcal{S}}$ with names in G to the subset A.

Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two pomsets. Put $E_i = E_{\mathcal{S}_i} \cup I_i$ for $i = 1, 2$. And let L_i be the language of the relation and function symbols interpreted by $F_{\mathcal{S}_i}$. Assume that for some A and G, $A = E_1 \cap E_2$ and $G = L_1 \cap L_2$, and the restrictions of $\mathcal{S}_1$ and of $\mathcal{S}_2$ to A and G are the same. Then the collection $(\mathcal{S}_1 \| \mathcal{S}_2) = S$ is defined to be the system of all pomsets $\mathcal{S}$ in the language $L_1 \cup L_2$ such that the restriction of $\mathcal{S}$ to E_i and L_i is $\mathcal{S}_i$ for $i = 1, 2$. It is not difficult to see that S is not empty (under the above assumption on the equality of the restrictions). Also, taking concurrent unions is a commutative operation.

Restrictions and concurrent unions can be used to define the semantics of communicating processes. As an example, suppose P_1 and P_2 are two processes that access some communication device D. Suppose D supports operations of type OP_1, executed in P_1, and OP_2 executed in P_2. (For example OP_1 is read, and OP_2 is write.) Let Y_1 and Y_2 be the systems describing P_1 and P_2 respectively, and let Y be the system describing D.

(A system is a collection of pomsets; it represents a process.) So if $\mathcal{A}$ is a pomset in Y_1, $E_{\mathcal{A}}$ contains events that are executions of OP_1 (and other events not connected to D). Similarly, pomsets in Y_2 contain events of type OP_2. The pomsets in Y contain events of both types, since the pomsets of Y describe the correct behavior of the device D.

It is possible that the pomsets in Y contain functions that are not in Y_1 or Y_2 but are needed for the specification of the device D. This shows the need to defined concurrent unions for pomsets with different languages.

Now if $\mathcal{A} \in Y_1,\ \mathcal{B} \in Y_2,\ \mathcal{C} \in Y$ and it happens that:

(1) The restrictions of $\mathcal{A}$ and of $\mathcal{C}$ to operation executions of OP_1 are the same.

(2) The restrictions of $\mathcal{B}$ and of $\mathcal{C}$ to operation executions of OP_2 are the same.

Then the collection $\mathcal{A}\|\mathcal{C}\|\mathcal{B} = S$ can be formed. Any $\mathcal{D} \in S$ is a pomset in which P_1 and P_2 communicate through D. The system of all such concurrent unions is denoted $Y_1\|Y\|Y_2$. This system represents the concurrent runs of P_1 communicating with P_2 through device D. In Section 5.3 we will use a different definition for $\mathcal{S}_1\|\mathcal{S}_2$ which is applicable in case $\mathcal{S}_1$ and $\mathcal{S}_2$ are states. In this case $E_i = E_{\mathcal{S}_i}$ is an antichain and then it is possible to define $\mathcal{S}_1\|\mathcal{S}_2$ not as a system but as a single pomset, by declaring $E_1 \cup E_2$ to be an antichain (after the suitable identifications are made).

5 Semantics

The aim of this part is to sketch a (denotational) semantics to a very simple language which nontheless allows concurrent executions of programs and recursive calls to procedures. The use of 'states' makes it almost as simple (or difficult) to deal with concurrent as with sequential programs. First we describe the language, then its semantics, and finally we say a little about the construction of domains (in Section 6).

5.1 A simple language

The language contains a set of basic instructions which consists of the following:

1. Communication instructions, such as $WRITE(R, x)$ or $GET(Q, x)$, where R and Q are communication devices and x is a data variable.
2. Assignment instructions. These have the form $x := \theta$, where x is a data variable and θ is a term built from the data variables with the aid of the the language functions (addition etc.).

3. Procedure calls. There is a collection **PVar** of Procedure identifiers (variables), and if $X \in \mathbf{PVar}$ then the statement 'X' is a basic instruction. (So procedures are called without parameters.)

These basic instructions can be composed into 'compound instructions' which are formed by the following operations:

1. Any basic instruction is also a compound instruction.
2. Concatenation: '(p;q)' where p and q are compound instructions.
3. Conditional instruction: '*IF* α *THEN* p' where α is a propositional statement involving data (and boolean) variables, logical connectives and the equality symbol. p is a compound instruction.

A 'sequential program' has the form

$$d_1; \ldots d_n; p$$

where the d_i's are procedure declarations (see below) and p is a compound instruction.

A 'procedure declaration' has the form

$$\textbf{procedure } X; p$$

where X is a procedure variable and p is a compound instruction. This procedure declaration binds the compound instruction p to the procedure name X.

Finally, a 'concurrent program' has the form

$$a_1 \| a_2 \ldots \| a_n$$

where the a_i's are sequential programs. For simplicity, we assume that the set of variables used by the a_i's are disjoint.

Obviously, we have swept aside some of the harder problems in language semantics. For example all our variables are globals, and procedures are parameterless. However, our aim is rather limited: only to point to the possibility of using the established tools of semantics for the description of concurrently communicating sequential programs (with the possibility of noninterleaving events).

5.2 Semantics

If the meaning of a program is its behavior (and not only the input/output relation) then the semantics should assign to a program fragment the collection of its possible behaviors (namely, a system of pomsets). It seems however more convenient to make explicit the dependency of the behavior on the (initial) environment, and let the denotation of a program construct be a map from the domain **Env**, of environments, to the domain **Sys** of all possible systems.

An environment was defined to be a map from the set of variables to the set of possible values. Let **Var** be the domain of variables of our language (process and data variables), and **Val** be the domain of possible values. So

$$\mathbf{Env} = \mathbf{Var} \to \mathbf{Val}.$$

In words, the domain of environments consists of the maps assigning values to the variables. There are two types of variables in our simple language: data and procedure variables, and accordingly:

$$\mathbf{Val} = \mathbf{Data} + \mathbf{Proc}.$$

(It is assumed that if $f \varepsilon \mathbf{Env}$, then for a data variable x, $f(x)$ is a data value, and for a procedure variable X, $f(X)$ is in **Proc**, the domain of process values.

The domain **Moment** contains all the pomset states (as defined in Definition 3.3). The term moment is used here rather than state, because in the context of denotational semantics 'state' has a different meaning.

Recall that in Section 3.5 the pomset concept was extended to include two environments (the second, only for terminating pomsets). Using the same intuition, let us define the domain **State** by

$$\mathbf{State} = \mathbf{Moment} \times \mathbf{Env} \times \mathbf{Env}.$$

In words, a state in **State** is a triple consisting of a pomset state and two environments—an initial and a final. If $T = (m, a, b)$ is in **State**, then m is called the pomset of T; a and b are the initial and final environments of T; the initial (final) pomset of T is defined to be the initial (final) pomset of m as defined in Section 3.3. The *initial part* of T is defined as the pair consisting of the initial pomset and the initial environment of T. Similarly, the final part of T consists of the final pomset and final environment of T.

It is useful to also have two special states, the **undefined** and the **null** state in **State**. The elements of any domain are partially ordered by the information they carry. For **State**, the **null** state is less informative than any other state; otherwise the states are incomparable. The use of the **undefined** state will be described later.

A pomset is represented here by a concatenated sequence of states: $T_1; T_2 \ldots$, where $T_i \in \mathbf{State}$ are such that the final part (pomset and environment) of T_i equals the initial part of T_{i+1}. A *system* is a collection of pomsets, it is the formal counterpart of the notion 'behavior'.

A simple way to represent a system is as a partial map $f : \omega^+ \to \mathbf{State}$. Here ω^+ is the discret domain of all finite non empty sequences of integers, and the domain (source) of f is assumed to be closed under initial segments. It is also assumed that if $\nu, \nu' \in \omega^+$ are in the domain of f and ν' is a successor of ν (i.e., $\nu' = \nu\hat{}\langle n \rangle$ for some n), then the final part of $f(\nu)$ equals the initial part of $f(\nu')$, unless $F(\nu)$ is the **null** state, and then we drop this requirement. In case $f(\nu) = \mathbf{null}$, for every extension ν' of ν, $f(\nu') = \mathbf{null}$.

Suppose that the sequence $\nu = \langle n_1 \dots n_l \rangle$ is in the domain of f and set $\nu_i = \langle n_1 \dots n_i \rangle$, then the sequence of states $f(\nu_1), \dots, f(\nu_l)$ represents a pomset—the concatenation of these states. We say that the resulting pomset is 'represented' in f by the branch determined by ν (the branch $\nu_1, \dots, \nu_l$). The branches of ω^+ (finite or infinite) thus *represent* the computations. I.e., we say that pomset $\mathcal{S}$ is *represented* in f (by ν) iff ν is a branch of the form $\nu = \{\nu_i | 1 \leq i < i_0\}$, where $\nu_i \in \omega^+$ is of length i and is in the domain of f, and the concatenation of the states $f(\nu_i)$'s gives $\mathcal{S}$ (i_0 can be finite or ω).

Actually, instead of dealing with partial maps, we shall take fully defined maps and allow the **undefined** state to be in the range (target) of f. We assume that if $f(\overline{n}) = \mathbf{undefined}$ for $\overline{n} \in \omega^+$, then, for every extension $\overline{m}$ of $\overline{n}$, $f(\overline{m}) = \mathbf{undefined}$. The *source* of f is then defined to be the set of $\overline{n}$'s such that $f(\overline{n}) \neq \mathbf{undefined}$.

A maximal node in the source of f is called a *leaf*: it is a node $\overline{n} \in \omega^+$ such that $f(\overline{n}) \neq \mathbf{undefined}$ but $f(\overline{m}) = \mathbf{undefined}$ for every extension $\overline{m}$ of $\overline{n}$. It is probably a technical point, but it seems that we need to know whether $\overline{n}$ is a leaf directly, and not only by checking that $f(\overline{m}) = \mathbf{undefined}$ for all the extensions of $\overline{n}$. For this, we postulate the existence of a special 'status variable' in **Var**, and assume that its value at the final environment of $f(\overline{n})$ indicates whether $\overline{n}$ is or is not a leaf of f. Thus if $f(\overline{n}) = \mathbf{null}$ then $\overline{n}$ *is* in the source of f, and it is not a leaf and for every extension $\overline{m}$ of $\overline{n}$, $f(\overline{m}) = \mathbf{null}$ as well. $f(\overline{n}) = \mathbf{null}$ convey lack of knowledge, which could change with further information; $f(\overline{n}) = \mathbf{undefined}$ is a definite knowledge that $\overline{n}$ is not in the source of f and will never be. The equation for the domain of systems is

$$\mathbf{Sys} = \omega^+ \to \mathbf{State}.$$

It is convenient to have also the following requirement on the domain **Sys**: *For every $f \in \mathbf{Sys}$ and $\nu \in \omega^+$, let $\mathcal{T}_0$ be the pomset represented in f by the branch that ν determines. Suppose the pomset $\mathcal{T}_1$ is represented in f (perhaps by a branch that does not extend ν), and $\mathcal{T}_1$ extends $\mathcal{T}_0$. Let $\mathcal{S}$ be a state such that the concatenation $\mathcal{T}_0;\mathcal{S}$ is extended by $\mathcal{T}_1$, then for some $n \in \omega$, $f(\nu\hat{}\langle n \rangle) = \mathcal{S}$.*

Thus, if we let Y denote the system of all pomsets represented by f as well as their initial segments, then this condition says that any extension of the pomset represented by ν that is still in Y can be obtained by extending

ν.

It follows that if $f \varepsilon \mathbf{Sys}$ then for any pomset, $\mathcal{A}$, represented in f, *all* the decompositions of $\mathcal{A}$ into sequences of states appear in f. This is needed to ensure that the definition of the concurrent execution $(p \parallel q)$ given below, results in *all* possible concurrent execution of p and q.

Now the domain of processes is defined by

$$\mathbf{Proc} = \mathbf{Env} \to \mathbf{Sys}.$$

In words, the domain of processes consists of the mappings from the environments into the systems, which are representing collections of pomsets. For $f \in \mathbf{Proc}$, $\sigma \in \mathbf{Env}$, and $\langle n \rangle \in \omega^+$, if $f(\sigma)\langle n \rangle$ is not the **null** or the **undefined** state, then the initial environment of $f(\sigma)\langle n \rangle$ is required to be σ. (Here, $\langle n \rangle$ is the sequence of length 1 containing n.)

We shall later need the undetermined (null) system $s_0 \in \mathbf{Sys}$ which is such that $s_0(\langle \overline{n} \rangle) = \mathbf{null}$, for every $\overline{n} \in \omega^+$. The undetermined process $p_0 \in \mathbf{Proc}$ is such that $p_0(\sigma) = s_0$ for every $\sigma \in \mathbf{Env}$.

To sum up: The recursive set of domain equations is the following. There are initial domains given by the language and its intended semantics: These are ω^+, **Var**, **Data** and **Moment**. Then **Val**, **Env**, **State**, **Sys** and **Proc** are constructed to satisfy the following:

1. $\mathbf{Val} = \mathbf{Data} + \mathbf{Proc}$.
2. $\mathbf{Env} = \mathbf{Var} \to \mathbf{Val}$.
3. $\mathbf{State} = \mathbf{Moment} \times \mathbf{Env} \times \mathbf{Env}$.
4. $\mathbf{Sys} = \omega^+ \to \mathbf{State}$.
5. $\mathbf{Proc} = \mathbf{Env} \to \mathbf{Sys}$.

As indicated, these equations must be refined. For example the domain **Sys** should be so defined that its elements give rise to sequences of states that can be concatenated—the final pomset and environment of a state in a sequence must be equal to the initial pomset and environment of the succeeding state.

5.3 Serial and concurrent executions

For p and q in **Proc**, we are going to define $(p;q) \in \mathbf{Proc}$ and $(p \parallel q) \in \mathbf{Proc}$ to be the serial and concurrent processes (respectively) executing p and q.

First the serial concatenated execution $r = (p;q) \ : \mathbf{Env} \to \mathbf{Sys}$ is defined. For each $\sigma \varepsilon \mathbf{Env}$ we must determine $r(\sigma) = h \ \varepsilon \ (\omega^+ \to \mathbf{State})$.

Given $\overline{n} = \langle n_1, \ldots, n_k \rangle \;\varepsilon\; \omega^+$, $h(\overline{n})$ is defined by the following. Let $p(\sigma) = f \;\varepsilon$ **Sys**. If $\overline{n}$ is in the source of f (i.e., $f(\overline{n}) \neq$ **undefined**), then put $h(\overline{n}) = f(\overline{n})$. If $\overline{n}$ is not in the source of f and it is not above a leaf of f, then $h(\overline{n}) =$ **undefined**. Otherwise, $\overline{n}$ is not in the source of f and is above a leaf of f. Let then $1 \leq i < k$ be the last index so that $l = \langle n_1, \ldots n_i \rangle$ is in the source of f. Decompose $\overline{n} = \overline{l}\overline{m}$, where $\overline{l}$ of length i is a leaf in the source of f. Let μ be the final environment of the state $f(\overline{l})$, and suppose the final pomset of $f(\overline{l})$ equals the initial pomset of $q(\mu)\langle n_{i+1} \rangle$. Then $h(\overline{n}) = q(\mu)\langle n_{i+1}, \ldots, n_k \rangle$. (Otherwise, $h(\overline{n})$ is **undefined**.)

In plain words, the process $p;q$ gives to each environment σ the system obtained by concatenating all possible sequences of states $S_1 \ldots S_i$, $S_{i+1} \ldots S_k$ where the first part forms a complete pomset in $p(\sigma)$ (i.e., not a proper initial segment) and the second part is obtained by looking at the environment obtained in the first part and applying q to it. Remark that a nonterminating execution in p (corresponding to an infinite branch of f) is also a nonterminating execution in $p;q$.

Let us turn now to the definition of parallel composition $p \parallel q$. The operator $\parallel$: **Proc**$\times$**Proc** $\to$ **Proc** depends on a restriction map, π, which is not explicitly shown for notational simplicity. A restriction is a map $\pi :$ **Moment** $\to$ **Moment** (see Section 4.2). We will first define the parallel composition of two states, and then the composition of two systems (elements of **Sys**), and finally the parallel composition of two processes (in **Proc**).

For $s_1, s_2 \in$ **State** $=$ **Moment**$\times$**Env**$\times$**Env**, where $s_i = (m_i, a_i, b_i)$ (m_i is the pomset state of s_i, and a_i, b_i are the initial and final environments of s_i). Assume that $\pi m_1 = \pi m_2 = p$. Then $m = (m_1 \parallel m_2)$ is formed by unifying the two antichains of m_1 and m_2 into a single antichain. For simplicity, assume that the set of variables of s_1 and s_2 are disjoint (i.e., the sources of the functions a_1, b_1 and a_2, b_2 are disjoint). So $a = a_1 \cup a_2$ and $b = b_1 \cup b_2$ can be defined with no conflicts. Now define

$$s_1 \parallel s_2 = (m, a, b).$$

Put $s_1 \parallel s_2 =$ **undefined**, in case m_1 and m_2 do not have the same projection. (Also, (**undefined** $\parallel s_2$) $=$ ($s_1 \parallel$ **undefined**) $=$ **undefined**.)

Suppose now S_1, $S_2 : \omega^+ \to$ **State** are two elements of **Sys**. We are going to define the concurrent union $S = S_1 \parallel S_2 \in$ **Sys**. For this, fix a bijection $c : \omega^+ \to \omega^+ \times \omega^+$ that is order and length preserving (the length of α, β and γ are all equal when $c(\alpha) = (\beta, \gamma)$). Given a finite node $\nu \in \omega^+$, let $c(\nu) = (\nu_1, \nu_2)$. Naturally, we wish to define $S(\nu)$ to be $S_1(\nu_1) \parallel S_2(\nu_2)$, and in case this cannot be done (if $\pi S_1(\nu_1) \neq \pi S_2(\nu_2)$) we set $S(\nu)$ to be **undefined**. Yet, we also want that if $S(\nu)$ is undefined then for every extension ν' of ν, $S(\nu')$ is undefined too. For this reason $S(\nu)$ is defined by induction on the length of ν:

If for some ν' below ν, $S(\nu')$ is **undefined**, then $S(\nu)$ is **undefined**. Otherwise, $S(\nu)$ is $S_1(\nu_1) \parallel S_2(\nu_2)$ (which is possibly **undefined**).

Now we define $r = (f \parallel g)$ for $f,\ g \in \mathbf{Proc}$. $r : \mathbf{Env} \to \mathbf{Sys}$. Given $\sigma \in \mathbf{Env}$, let $S_1 = f(\sigma),\ S_2 = g(\sigma)$; then $S_1,\ S_2 \in \mathbf{Sys}$ and we put $r(\sigma) = S_1 \parallel S_2$.

5.4 Denotations

For every construct, c, of the simple language, we will to define its denotation $\mathbf{C}\,(c) \in \mathbf{Proc}$. For that, $\mathbf{C}\,(c)(\sigma)\nu \in \mathbf{State}$ has to be defined for every $\sigma \in \mathbf{Env}$ and $\nu \in \omega^+$. It may be clearer to define the collectrion of pomsets that are to be represented by $\mathbf{C}\,(c)(\sigma)$ rather than to explicitly show the tree of states. We will outline the definition of $\mathbf{C}\,(c)$ for only five cases. We begin with the denotation of a concurrent program assuming its serial processes already have a denotation.

1. Let c be a concurrent program of the form $a_1 \parallel \ldots \parallel a_n$. The denotation $\mathbf{C}\,(c)$ depends on the semantics of the communication devices used by the processes. As an example, suppose a queue of capacity k, as specified in Section 3.4, is used by a_1 (the producer) and by the two consumers a_2 and a_3. So $c = a_1 \parallel a_2 \parallel a_3$ is a program in which the three processes communicate through the queue. Given $\sigma \in \mathbf{Env}$, set $S_i = \mathbf{C}\,(a_i)\sigma$ for $i = 1, 2, 3$. Then S_1 is the system describing all the possible executions of the producer's program, and likewise S_2 and S_3 describe the consumer. But there is nothing in S_1, S_2, and S_3 to indicate that a queue is used. So, for example, S_1 may contain an execution that could never be normally observed in a concurrent execution of c. It is only by specifying the communication device that the denotation $\mathbf{C}\,(c)$ can be defined, and there should be enough freedom to enable any possible specification. So let $Q : \omega^+ \to \mathbf{State}$ be the system that describes all the pomsets, $\mathcal{S}$, that satisfy the queue specifications described in Section 3.4. Now define

$$\mathbf{C}\,(c)\sigma = ((Q \parallel S_1) \parallel S_2) \parallel S_3$$

The definition of $\parallel$ was given in Section 5.3. Intuitively, $Q \parallel S_1$, for example, is the system of all pomsets that can be obtained by running in parallel a pomset $\mathcal{S}$ in Q and a pomset $\mathcal{S}_1$ in S_1 when the projections of $\mathcal{S}$ and $\mathcal{S}_1$ on the enqueue events are equal, and by identifying these common events. So the pomsets represented in $\mathbf{C}\,(c)\sigma$ describe the executions of the program c that use a queue as specified.

2. Take now, for example, a basic communication command such as $c =$ '$GET(Q, x)$' where Q is a name of a queue and x is a data variable.

To define $\mathbf{C}\,(c)(\sigma)$ for $\sigma \in \mathbf{Env}$, we look at the set U of all states $(m, e_1, e_2) \in \mathbf{State}$ such that $e_1 = \sigma$ and e_2 equals e_1 except perhaps at the variable x, and m is a moment with a single event in E_m which is a dequeue event of value $e_2(x)$. The system representing all the pomsets in U is defined to be $\mathbf{C}\,(c)(\sigma)$. Observe that there is nothing to indicate that a queue is used rather than any other device, and all possible outcomes of the GET are represented.

3. For a procedure call X, $\mathbf{C}\,(X)$ denotes the activation of the procedure named X. So, for any $\sigma \in \mathbf{Env}$, $\mathbf{C}\,(X)\sigma = (\sigma(X))\sigma$. In words, for a procedure variable $X \in \mathbf{PVar}$, the denotation, $\mathbf{C}\,(X)$, of the activation of X is the element of $\mathbf{Proc}$ that assigns to every environment σ the system S defined as follows: Look at the value $p = \sigma(X)$ of X in the environment σ, then $p \in \mathbf{Proc} = \mathbf{Env} \to \mathbf{Sys}$, and we define $S = p(\sigma)$.

4. For sequential composition, $(c_1; c_2)$,

$$\mathbf{C}\,(c_1; c_2) = \mathbf{C}\,(c_1); \mathbf{C}\,(c_2).$$

Of course the symbol ';' is used here in two distinct ways: as a syntactic symbol and as defined in Section 5.3.

5. The declaration, (**procedure** $I; p$), which binds procedure variable I to body p, is somewhat like an assignment command in that it only changes the environment, but it is complicated of course by the possibility of recursion (because p may use its own name I, directly or indirectly), and its denotation involves the use of fixed points. The function $\mathit{fix} : (\mathbf{Proc} \to \mathbf{Proc}) \to \mathbf{Proc}$ gives to every function $f : \mathbf{Proc} \to \mathbf{Proc}$ a 'minimal fixed point', $p = \mathit{fix}(f)$, such that $f(p) = p$. To obtain p, start with the undetermined process p_0, iteratively apply f to it, get the increasing sequence $f^i(p_0)$, $i = 0, 1, \ldots,$ and take its supremum to be $\mathit{fix}(f)$.

We need the substitution function $sub_f^X : \mathbf{Env} \to \mathbf{Env}$ which changes the environment σ by assigning the value $f \in \mathbf{Proc}$ to the variable $X \in \mathbf{PVar}$. So

$$sub_f^X(\sigma)(Y) = \begin{cases} f & \text{if } Y = X; \\ \sigma(Y) & \text{otherwise.} \end{cases}$$

Now let $I \in \mathbf{PVar}$ be a fixed identifier, p a compound instruction. Define

$$t_p^I : \mathbf{Proc} \to \mathbf{Proc}$$

by

$$t_p^I(f) = \mathbf{C}\,(p) \circ sub_f^I$$

($\circ$ is function composition). Intuitively, $t_p^I(f)$ is the denotation of p when the variable I is interpreted as the name of procedure f. So $\mathit{fix}(t_p^I)$ is 'the denotation of I after the declaration that binds I with body p'.

The denotation of a procedure declaration can now be described in accordance with this intuition, define $\mathbf{C}\,(\mathbf{procedure}\ I; p) = d$ as follows. $d : \mathbf{Env} \to \mathbf{Sys}$, is the function that takes any $\sigma \in \mathbf{Env}$ to the system $d(\sigma) = S \in \mathbf{Sys}$ that has the following property. S comprises of all representations of the pomset $(m, \sigma, sub^I_{\mathit{fix}(t_p^I)}(\sigma))$ which is a state where $m \in \mathbf{Moment}$ contains a single event, a 'declaration', which, intuitively, only changes the environment from σ to $sub^I_{\mathit{fix}(t_p^I)}(\sigma)$.

6 On domains

This section describes a modified version of Scott's [1982] definition of 'information system'. The modification consists in dropping the entailment relation, and showing that there always exists a definable (maximal) entailment relation which could be used instead.

Information systems were defined by Scott [1982] and used to provide domains for specifying the semantics of computer languages. An 'information system' is a structure $A = (D_A, \Delta_A, Con_A, \vdash)$ where D_A is a set; $\Delta_A \in D_A$ is the 'null' element; Con is a collection of finite subsets of D_A (the 'consistent' sets); and $\vdash$ is a relation on Con. Intuitively, an information system capture the idea that we often only have partial information on the objects with which we deal (the 'elements' of the system), but still want to be able to operate on these partially specified objects. The sets in Con are thought of as finite partial information, and $\vdash$ is an entailment relation. Accordingly, an 'element' of the information system is a subset $x \subseteq D$ such that (1) every finite subset of x is in Con, and (2) if $E \subseteq x$ and $E \vdash F$, then $F \subset x$.

Scott also shows how information systems can be combined to form new information systems. The most interesting combination is the one by which the information system $A \to B$ (the 'function space') is formed. It seems to us that in some situations the entailment relation is not naturally defined (that is, it is not closely related with the objects studied). Especially in the definition of the function space there does not seem to be a natural explanation for the particular choice of the entailment relation (except for the fact that it leads to a smooth theory), and we try here to see how far the theory of information systems can be developed without the entailment relation— or, more precisely, with the entailment relation that is always present, namely $A \vdash B$ iff it is impossible for further information added to A to contradict B.

We continue to use the same term 'information system' for our systems

which are devoid of $\vdash$, because we will refer here only to such systems.
Definition 6.1 An 'information system' is a triple $A = (D_A, \Delta_A, Con_A)$

where D_A is a set (here finite or countable), $\Delta_A \in D_A$ is a distinguished element, and Con is a collection of finite subsets of D_A such that the following holds:

1. Con_A contains all singletons $\{a\}$, for $a \in D_A$, and is subset closed (i.e., if $X \subset Y \in Con_A$, then $X \in Con_A$).

2. If $X \in Con_A$, then $X \cup \{\Delta_A\} \in Con_A$.

Given an information system, A, let us define the '(maximal) entailment relation', $\vdash_A$, on Con_A: For $u,\ v \in Con_A$,

$$u \vdash_A v \textit{ iff for every } x, u \cup x \in Con_A \textit{ implies } v \cup x \in Con_A.$$

The following obvious properties of entailment hold:

1. If $u \vdash_A v$ then $u \cup v \in Con_A$, and for every $x \supseteq u$ in Con_A, $x \vdash_A v$.

2. $u \vdash_A \{\Delta_A\}$.

3. $u \vdash_A v$ iff for every $a \in v$, $u \vdash_A \{a\}$.

4. $u \vdash_A v$ for every $v \subseteq u$.

5. $u \vdash_A v$ and $v \vdash_A t$ imply $u \vdash_A t$.

It follows now that for any information system A as above, $(A, \vdash_A)$ is an information system in the original sense of Scott's definition. Thus our discussion may be viewed as a remark on a particular class of Scott's information systems.

We shall say that $e \subset D_A$ is 'consistent' if, for every finite $E \subseteq e$, $E \in Con_A$. Two (or more) subsets, E and F, of D_A are said to be consistent if $E \cup F$ is consistent.

The *elements* of an information system A are defined to be the consistent and entailment closed subsets of D_A. That is $x \subset D_A$ is an 'element' of A iff every finite subset of x is in Con_A, and whenever $E \subset x$ and $E \vdash_A F$, then $F \subset x$ as well. The set of elements of A is denoted by $\mid A \mid$. A maximal element is one that is not included in any other element.

Given two information systems, A and B, the disjoint sum, $A + B$, the product, $A \times B$, and the function space, $A \to B$, are defined as in Scott [1982]—but now there is no need to define the entailment relation for each construct. We will restate the definition of the function space, in order to prove that the (modified) definition of 'approximable mappings' gives exactly the elements of the space. For that, let us first introduce some notations.

For any $F \subseteq X \times Y$ let us define $left(F) = \{x \mid \exists y\ (x,y) \in F\}$. Similarly, $right(F)$ is defined, and, for any W, $right_W(F) = \{y \mid \exists x \subseteq W$ such that $(x,y) \in F\}$. $\cup left(F)$ is called the left projection of F. It is the set of all a such that $a \in x$ for some $(x,y) \in F$. Similarly, $\cup right(F)$ is called the right projection of F. For any W, $\cup right_W(F)$ is denoted F_W.

For two information systems, A and B, define the function space $C = A \to B$ as the following information system:

1. $D_C = Con_A \times Con_B = \{(u,v) \mid u \in Con_A \ \&\ v \in Con_B\}$.

2. $\Delta_C = (\emptyset, \emptyset)$.

3. For finite $E \subset D_C$,

$$E \in Con_C \quad \text{iff} \quad \textit{for every} \quad X \in Con_A,\ E_X \in Con_B.$$

Equivalently,

$$E \in Con_C \quad \text{iff} \quad \textit{for every } E_0 \subset E, \textit{ if the left projection of } E_0 \textit{ is in } Con_A, \textit{ then the right projection of } E_0 \textit{ is in } Con_B.$$

To define *approximable mappings*, we first need the following generalization of the entailment notation. Given an information system, A, define for $u, u_1, \ldots, u_n \in Con_A$
$u \vdash_A (u_1 \vee \ldots \vee u_n)$ iff *For every* $x \supset u$ *in* Con_A *there is* $i \leq n$ *such that* $x \cup u_i \in Con_A$.

Definition 6.2 Let A and B be information systems. An *approximable mapping*, $f : A \to B$, is a binary relation $f \subset Con_A \times Con_B$ such that

1. $\emptyset\ f\ \emptyset$.

2. $ufv \ \&\ ufv' \Rightarrow uf(v \cup v')$.

3. $u' \vdash_A u,\ ufv,\ v \vdash_B v' \Rightarrow u'fv'$.

4. $u \vdash_A u_1 \vee \ldots \vee u_n,\ (\forall i\ u_i f v) \Rightarrow ufv$.

Remark: It follows that if F is a finite subset of an approximable mapping $f : A \to B$ then:

1. If $X \in Con_A$, then $F_X \in Con_B$. (Hence f is a consistent subset of $A \to B$.)

2. If $u \in Con_A$ is such that $x \subseteq u$ whenever $(x,y) \in F$, then

$$(u, right(F)) \in f.$$

Theorem 6.3 *The approximable mappings $f : A \to B$ are exactly the elements $f \in\mid A \to B \mid$.*

Proof: Suppose first that $f \in\mid A \to B \mid$. Then $f \subset D_{A\to B}$ is consistent and entailment closed. So $f \subset Con_A \times Con_B$. Since f is entailment closed, and as $\emptyset \vdash \{\Delta\}$ is always true, $\Delta_{A\to B} \in f$. That is, $\emptyset f \emptyset$. Suppose now that $(u, v),\ (u, v') \in f$; we are going to show that $(u, v \cup v') \in f$. Since f is consistent, $\{(u, v),\ (u, v')\} \in Con_{A\to B}$, and hence $w = v \cup v' \in Con_B$. To prove ufw, it suffices to observe that

$$\{(u, v),\ (u, v')\} \vdash_{A\to B} \{(u, w)\},$$

and then to use the fact that f is entailment closed. To prove items 3 and 4 in the definition of an approximable mapping, suppose that

$$u \vdash_A u_1 \vee \ldots \vee u_n,\ (\forall i\ u_i f v),\ \text{and}\ v \vdash_B v'.$$

We have to prove that ufv'. For this, one proves that

$$\{(u_1, v), \ldots, (u_n, v)\} \vdash_{A\to B} \{(u, v')\},$$

and uses the fact that f is entailment closed.

For the second direction of the proof we make some preliminary definitions and a lemma (which will also be used later). Let U be an information system and $L \subset D_U$ (not necessarily consistent). $T \in Con_U$ is said to be 'maximal on L' iff for every consistent $T' \supset T$, $T \cap L = T' \cap L$. In other words, T is maximal on L if T is consistent, but for every $a \in L \setminus T$, $T \cup \{a\}$ is inconsistent.

The collection of all $T \in Con_U$ that are maximal on L is denoted $M_U(L)$. The collection of all $T \in Con_U$ that are maximal on L and such that $L_0 = T \cap L$ is denoted $M_U(L, L_0)$.

Observe that for finite $L \subset D_U$, and $B \in Con_U$,

$$B \vdash_U \bigvee\{L \cap T \mid T \in M_U(L) \& B \subseteq T\}.$$

(This disjunction is finite since L is finite.)

Lemma 6.4 *Let A and B be information systems and $M = (A \to B)$. If $F \in Con_M$ and $(a, b) \in D_M$, then the following are equivalent:*

1. *$F \not\vdash_M \{(a, b)\}$.*

2. *Put $L = \bigcup left(F)$. There exists $w \supseteq a$ in Con_A, maximal on L, such that $F_w \not\vdash_B b$. (Recall that $F_w = \bigcup right_w(F)$.)*

Proof: Assume first *1*. So there is $G \in Con_M$ such that $F \subseteq G$ and $E = G \cup \{(a,b)\}$ is not consistent. Thus there is $X \in Con_A$ such that $E_X \notin Con_B$. Since G is consistent, it follows that $a \subseteq X$. Now extend X and find $X \subseteq w$ that is maximal on L. Then $E_X \subseteq E_w$ and so $E_w \notin Con_B$. w is as required since $F_w \not\vdash_B b$ because $F_w \subseteq G_w \in Con_B$ and $G_w \cup b = E_w$.

Assume *2* and let us check that *1* holds. Since $F_w \not\vdash_B b$, there is $z \supseteq F_w$ in Con_B such that $z \cup b$ is not in Con_B. Hence $\{(w,z),(a,b)\}$ is not in Con_M. But $F' = F \cup \{(w,z)\}$ is consistent, and hence *1*. To see that F' is in Con_M, suppose that $X \in Con_A$, and let us check that $F'_X \in Con_B$. Since F is consistent we may assume that $w \subseteq X$, and then $F'_X = F_X \cup z$. Since w is maximal on L, for every $(x,y) \in F$, if x is consistent with w then $x \subseteq w$, and in such a case $y \subseteq F_w \subseteq z$. Thus $F_X \subseteq z$ and $F'_X = z$.

Returning to the proof of Theorem 6.3, assume $f : A \to B$ is an approximable mapping, and we will show that $f \in \mid A \to B \mid$. By Definition 6.2, $f \subset D_{A \to B}$. We remarked that f is consistent (after Definition 6.2). To prove that f is entailment closed, suppose $F \subset f$ is finite, and $F \vdash_{A \to B} \{(u,v)\}$; we will show that $(u,v) \in f$.

Put $L = \bigcup \mathit{left}(F)$. For every t, set $L_t = t \cap L$. Then, since any consistent t can be extended to be maximal on L,

$$u \vdash_A \bigvee \{L_t \mid u \subseteq t \textit{ and } t \textit{ is maximal on } L\}.$$

For any $E \in Con_A$, $(E, F_E) \in f$, because $F \vdash_{A \to B} \{(E, F_E)\}$. Thus $(L_t, F_{L_t}) \in f$. If we show that $F_{L_t} \vdash_B v$, for every such t, then by the closure property 4 of Definition 6.2, $(u,v) \in f$.

By Lemma 6.4, since $F \vdash_{A \to B} \{(u,v)\}$, for every $t \supseteq u$ maximal on L, $F_t \vdash_B v$. But $F_t = F_{L_t}$, and hence $F_{L_t} \vdash_B v$ for each $t \supset u$ maximal on L. Hence $(u,v) \in f$. This ends the proof of Theorem 6.3.

To solve a recursive system of information-system equations, one may proceed in defining better and better approximations to the solutions and then take their unions. (For more details read Larsen and G. Winskel [1984].) We shall define a relation $U \lhd V$ on information systems (which indicates that V is a better approximation than U), and then show how the diverse constructions are preserved, e.g., if $U_1 \lhd U_2$ and $V_1 \lhd V_2$, then $(U_1 \to V_1) \lhd (U_2 \to V_2)$.

Say that $X \in Con_U$ is *full* iff $X \vdash_U Y$ for all consistent $Y \supset X$. Equivalently, for every two consistent sets, Y_1 and Y_2 extending X, $Y_1 \cup Y_2$ is consistent.

Definition 6.5 Let U and V be information systems. $U \lhd V$ iff

1. $D_U \subseteq D_V$.

2. For every finite $E \subseteq D_U$, $E \in Con_U$ iff $E \in Con_V$.

3. If $A \in Con_U$, then A is full in V if A is full in U.

4. For finite $X,\ Y_1, \ldots, Y_n \subseteq D_U$, if $X \vdash_U \bigvee\{Y_i \mid 1 \leq i \leq n\}$, then $X \vdash_V \bigvee\{Y_i \mid 1 \leq i \leq n\}$ as well.

Let us make the following remarks on this definition.

1. In applications, condition $D_U \subseteq D_V$ is replaced by the existence of an appropriate embedding $e : D_U \to D_V$. For example, if N is the discrete information system of the natural numbers, then $N \lhd (N \to N)$ by the embedding taking k to $(\{\Delta\}, \{k\})$.

2. Condition 3 is natural: Usually it is not the case that the closure under entailment (in V) of a maximal element of U is maximal in V, but if a finite set in U generates a maximal element (i.e., it is full) then the element it generates in V is maximal as well. The other direction in 3 is evident: If $A \in Con_U$ is full in V then it is full in U.

3. Assuming the first two conditions, condition 4 is equivalent to the "reflection property" which is the following: For every finite $A \subseteq D_U$ (not necessarily in Con_U), for every $B \in Con_V$, if B is maximal on A, then there exists $B' \in Con_U$ maximal on A and such that $B \cap A = B' \cap A$. We say in this case that B' *reflects* B on A.

(Let us check that 4 implies this reflection property. Assume 4. Since

$$B \cap A \vdash_U \bigvee\{A \cap T | T \in M_U(A)\ \&\ B \cap A \subseteq T\},$$

$$B \cap A \vdash_V \bigvee\{A \cap T | T \in M_U(A)\ \&\ B \cap A \subseteq T\}.$$

Hence B is consistent with some $A \cap T$ for $T \in M_U(A)$ such that $B \cap A \subseteq T$. By maximality, $B \cap A = T \cap A$.)

For $A \subseteq D_U$ and $A_0 \in Con_U$, *such that* $A_0 \subseteq A$, define the *width* of the configuration (A, A_0) to be the supremum of the sizes of families $\mathcal{A} \subseteq M_U(A, A_0)$ consisting of pairwise inconsistent members (such a family is called an 'antichain' in $M_U(A, A_0)$). The width of a configuration is ≥ 1 when $M_U(A, A_0) \neq \emptyset$ and it can be ω.

We claim the following: For $0 < k < \infty$, the width of (A, A_0) is k iff there is in $M_U(A, A_0)$ a maximal antichain of size k consisting of full sets. Indeed, suppose that the width is k and let $B_1, \ldots, B_k$ be an antichain in $M_U(A, A_0)$ (there are no antichains of size $k+1$ there); then all the B_i's are full or else k can be increased.

On the other hand, if $B_1, \ldots, B_k$ is a maximal antichain in $M_U(A, A_0)$ (there are no antichains there extending this one) and each B_i is full, then $M_U(A, A_0)$ contains no antichain of size $k+1$. To see this, suppose that

$X_1, \ldots, X_{k+1}$ is an antichain in $M_U(A, A_0)$, and we will get a contradiction. If some X_j is inconsistent with each B_i, then the antichain of the B_i's is not maximal. Hence every X_j is consistent with some B_i, and so there are two X_j's consistent with the same B_i which is thence not full.

A similar argument shows that if the width of (A, A_0) is ω, then there is in $M_U(A, A_0)$ an infinite antichain (the supremum is obtained). This is proved by contradiction. If all the antichains in $M_U(A, A_0)$ are finite, construct by induction a tree in $M_U(A, A_0)$ a follows. Suppose that the n-th level of the tree is a maximal antichain in $M_U(A, A_0)$, extend each node to a maximal antichain of sets in $M_U(A, A_0)$ that extend this node. Now the construction will not end at a finite stage or else the leaves form a maximal antichain of full sets and this implies a finite width. Hence the tree has an infinite branch and the infinite antichain is obtained by the splittings along this branch.

These considerations lead to the following lemma.

Lemma 6.6 *If $U \lhd V$, then the width of (A, A_0) in U (for $A \subseteq D_U$) is the same as its width in V. Hence for any finite $A \subset D_U$, and antichain $X \subset M_V(A)$, there is a map $\rho : X \to M_U(A)$ such that $\rho(v)$ reflects v on A, and, for distinct v_1 and v_2 in X, $\rho(v_1)$ and $\rho(v_2)$ are inconsistent.*

Proof: For the first part of the lemma, suppose the width of (A, A_0) in U is k, and let $B_1, \ldots, B_k$ be a maximal antichain of full sets in $M_U(A, A_0)$. By the reflection property, this family is maximal in V as well, and hence the width of (A, A_0) in V is k. If the width of (A, A_0) in U is ω, then it is obviously ω in V.

To prove the second part of the lemma, let $M = \{v \cap A \mid v \in X\}$. For each $A_0 \in M$ there may be several (say k many, where k is finite or ω) v's in X such that $v \cap A = A_0$. However, since the width of (A, A_0) in V is not greater than its width in U, there is in $M_U(A, A_0)$ an antichain of size k, and hence ρ can be defined. It is defined separately for each $A_0 \in M$ on $\{v \in X \mid v \cap A = A_0\}$ and the union of these functions is the desired ρ (observe that if $X, Y \in M_U(A)$ and $X \cap A \neq Y \cap A$ then X and Y are inconsistent). Now comes the main Lemma, for which $\lhd$ was introduced.

Lemma 6.7 *If $U \lhd V$ and $S \lhd T$, then $(U \to S) \lhd (V \to T)$.*

Proof: Set $L = (U \to S)$, $M = (V \to T)$. Let us check the four properties in Definition 6.5 needed to show that $L \lhd M$.

The inclusion $D_L \subseteq D_M$, and the equality of Con_L with Con_M on subsets of D_L are clear.

Suppose that $A \in Con_L$ is not full in M; we will show then that it is not full in L. So there is $(v, t) \in D_M$ compatible with A (i.e., its singleton is consistent with A) and such that

$$A \not\vdash_M (v, t).$$

We will provide $(v',t') \in D_L$ compatible with A such that $A \not\vdash_L \{(v',t')\}$.

Put $E = \bigcup \mathit{left}(A)$. By Lemma 6.4 there is $w \supseteq v$ in Con_V, maximal on E, such that for $y = \bigcup \mathit{right}_w(A)$, $y \not\vdash_T t$. By reflection, there is $w' \in Con_U$ that reflects w on E. So, w' is maximal on E, and $w' \cap E = w \cap E$. Thus $y = A_{w'}$.

Now, $y \cup t \in Con_T$ since A and (v,t) are compatible. So y is not full in T. Hence y is not full in S, and there is $t' \in Con_S$ such that t' is consistent with y but $y \not\vdash_S t'$. Hence A and $\{(w',t')\}$ are consistent in L, but $A \not\vdash \{(w',t')\}$ (by Lemma 6.4).

Finally, we prove reflection of M in L. Let $A \subseteq D_L$ be finite, and $B \in Con_M$ be maximal on A. Put $A_0 = B \cap A$. We shall find $B' \in Con_L$ maximal on A such that $A \cap B' = A_0$.

Let $E = \bigcup \mathit{left}(A)$, $F = \bigcup \mathit{left}(A \cup B)$. Let us define

$$K = \{v \mid v \in Con_V \text{ is maximal such that } v \subseteq F\}.$$

Then K is an antichain in $M_V(E)$. Fix a map ρ that assigns to every $v \in K$ some $\rho(v) \in Con_U$ that reflects v on E, and such that for distinct v's the values of ρ are inconsistent (by Lemma 6.6).

For each $v \in K$ let $t = B_v$ and let $\bar{t} \supseteq t$ be some extension in Con_T that is maximal on $G = \mathit{right}(A)$. Then let $\gamma(v) \in M_S(G, G \cap \bar{t})$ be some reflection of $\bar{t}$. Since $A_0 \subseteq B$, $(A_0)_v \subseteq t$. As $(A_0)_v \subseteq G$,

$$(A_0)_v \subseteq \gamma(v).$$

Define $B_1 \subset D_L$ by $B_1 = \{(\rho(v), \gamma(v)) \mid v \in K\}$. Now set $B' = A_0 \cup B_1$. We are going to prove that $B' \in Con_L$ and that for every $(u,s) \in A \setminus A_0$, $\{(u,s)\} \cup B'$ is not consistent.

To prove that $B' \in Con_L$, let $X \in Con_U$ be given, and we shall see that $B'_X \in Con_S$. $B'_X = (A_0)_X \cup (B_1)_X$. As $A_0 \in Con_L$ we may assume that $(B_1)_X \neq \emptyset$. But since $\{\rho(v) \mid v \in K\}$ is an antichain, there is then exactly one $v \in K$ with $\rho(v) \subseteq X$. So $B'_X = (A_0)_X \cup \gamma(v)$. But $(A_0)_X = (A_0)_v$. (Because $(A_0)_X \supset (A_0)_v$ follows from $v \cap E = \rho(v) \cap E \subseteq X$. For the second inclusion, take any $(a,b) \in A_0$ such that $a \subseteq X$. Then a is consistent with $\rho(v)$ and hence with v; by the maximality of $v \in K$, $a \subseteq v$.) We observed above that $(A_0)_X \subseteq \gamma(v)$, and thus $B'_X = \gamma(v) \in Con_S$.

We now prove the maximality of B' on A. Take any $(u,s) \in A \setminus A_0$. By its maximality, B is not compatible with (u,s), and hence there is $v \subseteq F$ such that $u \subseteq v$ and s is not consistent with B_v. We may assume $v \in K$. To prove that B' is not compatible with (u,s), take $(\rho(v), \gamma(v)) \in B'$ and prove that it is not compatible with (u,s). Indeed $u \subseteq \rho(v)$, and s is not consistent with $\gamma(v)$ because $\gamma(v)$ reflects some $\bar{t} \supseteq B_v$ on G.

7 Conclusion

The concept 'pomset with functions' presented here combines the system executions of Lamport [1986] and the pomsets of Gischer [1989] and Pratt [1986]. (It is unfortunate that divergent terminologies have evolved, since different aspects of the same concept were illuminated by these papers.

We are not the first to expand system executions. It is quite clear to any reader of Lamport [1986] that, at least informally, he enriches his structures by speaking about values obtained by reads etc. It is also clear that Gaifman [1989] was aware of the implications of adding the axioms of interval orders to pomsets. Ben-David [1988] has explicitly added unary predicates to the structures of Lamport, and thus combined the pomset notion with that of 'pure' system-executions.

The notion of a pomset is further extended by allowing functions (of bounded depth). One of the reasons for introducing these functions lies in the need to view systems in a hierarchical manner. An important benefit of the partial-ordering modeling approach—as opposed to the interleaving of atomic events—is the ability to consider higher-level views of system executions (for a definition see Lamport [1986]). In these higher-level views, some minute events are lumped together to form a single entity. In this passage from the concrete to the abstract, necessarily (and fortunately) some details are lost, and in order to preserve those lower-level properties that are important in the higher-level view, functions (or binary predicates) are used.

The notion of state was defined as a pomset that describes a moment (a pomset with no temporal relations). This was used to show the duality and equivalence of the state based and the event based approaches to concurrency.

A natural alternative approach is to define a state as an equivalence class of runs where two runs are equivalent iff they have exactly the same continuations. This idea is investigated in Probst and Li [1990] and it expresses the intuition that a state is the manifold of all future possibilities. We have chosen a different, more concrete, path in our definition, and one of our motives in this choice was the desire to show that states are finite objects. (However, our approach can be adapted to infinite domains as well; this is needed for denotational semantics, because then the realm of finite objects must be abandoned.)

We have outlined a possible approach to semantics of concurrency, which uses the notion of a pomset state. One advantage of our approach to semantics is the flexibility gained by having in one framework both the event approach and the state approach. Thus, the specifications for the communication devices are primarily given in terms of pomsets (this is an event approach), but the execution of sequential programs results (essentially) in a tree of states. When combined, the semantics of communicating processes can be clearly exposed—at least for the very simple language discussed here.

References

U. Abraham, S. Ben-David, and M. Magidor

1990 *On Global-Time and Inter-process Communication.* Semantics for Concurrency, Proceedings of the International BCS-FACS Workshop, pp. 311-323.

F. Anger

1989 *On Lamport's interprocess communication.* ACM Transactions on Programming Languages and Systems, Vol. 11 No. 3, pp. 404-417.

S. Ben-David

1988 *The global-time assumption and semantics for distributed systems.* 7th ACM Symp. on Principles of Distributed Computing, pp. 223-232.

P. C. Fishburn

1986 *Interval orders and interval graphs.* Wiley, New-York (Wiley-Interscience series in discrete mathematics).

H. Gaifman

1989 *Modeling concurrency by partial orders and non-linear transition systems.* In Logic and Models for Concurrency, edited by J. de Bakker, W. de Roever, and G. Rozenberg. Lecture Notes in Computer Science 354. Springer-Verlag, Berlin, pp. 467-488.

J. Gischer

1989 *The equational theory of pomsets.* Theoretical Computer Science, Vol. 61, pp. 199-224.

L. Lamport

1986 *On Interprocess Communication, Part I: Basic formalism, Part II: Algorithms.* Distributed Computing, Vol. 1, pp. 77-101.

K. G. Larsen and G. Winskel

1984 *Using information systems to solve recursive equations effectively.* Semantics of Data Types. Edited by G. Kahan et al. Lecture Notes in Computer Science 173. Springer-Verlag.

V. Pratt

1986 *Modeling Concurrency with Partial Orders.* International J. of Parallel Programming, Vol. 15, pp. 33-71.

D. K. Probst and H. F. Li

1990 *Modelling Reactive Hardware Processes Using Partial Orders.* Semantics for Concurrency, Proceedings of the International BCS-FACS Workshop, pp. 324-343.

D. S. Scott

1982 *Domains for Denotational Semantics.* In Automata, Lang., and Prog. Edited by M. Nielsen and E. M. Schmidt, Lecture Notes in Computer Science 140. Springer-Verlag, Berlin.

N. Wiener

1914 *A contribution to the theory of relative position.* Proc. Camb. Philos. Soc. 17, pp. 441-449.