Modelling Concurrency with Partial Orders*

Vaughan Pratt
Stanford University
Stanford, CA 94305
Pratt@SU-NAVAJO.ARPA

* Revision of "Some Constructions for Order-Theoretic Models of Concurrency"(1).

To appear in the International Journal of Parallel Programming, **15**, *1, c. Nov. 1986.*

Abstract

Concurrency has been expressed variously in terms of formal languages (typically via the shuffle operator), partial orders, and temporal logic, *inter alia.* In this paper we extract from these three approaches a single hybrid approach having a rich language that mixes algebra and logic and having a natural class of models of concurrent processes. The heart of the approach is a notion of partial string derived from the view of a string as a linearly ordered multiset by relaxing the linearity constraint, thereby permitting partially ordered multisets or *pomsets.* Just as sets of strings form languages, so do sets of pomsets form processes. We introduce a number of operations useful for specifying concurrent processes and demonstrate their utility on some basic examples. Although none of the operations is particularly oriented to nets it is nevertheless possible to use them to express processes constructed as a net of subprocesses, and more generally as a system consisting of components. The general benefits of the approach are that it is conceptually straightforward, involves fewer artificial constructs than many competing models of concurrency, yet is applicable to a considerably wider range of types of systems, including systems with buses and ethernets, analog systems, and real-time systems.

Keywords: concurrency, partial order, pomset, temporal logic, computation model.

§1 Introduction

1.1 An Application

In September 1983 at Cambridge University, England, STL (Standard Telecommunications Laboratories Ltd) and SERC (Science and Engineering Research Council) jointly sponsored a workshop on the analysis of concurrent systems(2). The workshop was organized around ten problems for solution by the attendees. Each problem informally described a concurrent system to be formally specified.

The first problem was probably the easiest. For whatever reasons, it also attracted the lion's share of the attention. The formal solutions presented varied in length from seven axioms of one line each (Koymans-de Roever) to two or more pages. The following paragraph reproduces verbatim the statement of the problem.

The "channel" between endpoints "a" and "b" can pass messages in both directions simultaneously, until it receives a "disconnect" message from one end, after which it neither *delivers nor accepts messages at that end. It continues to deliver and accept messages at the other end until the "disconnect" message arrives, after which it can do nothing. The order of messages sent in a given direction is preserved.*

The solutions that appear in the workshop proceedings[2] are complex out of all proportion to the intuitive complexity of the concept. The methods of this paper permit the following far shorter solution.

$$\pi\underline{\lambda}(((M^* \times A) \times AB)\|((M^* \times B) \times BA)) \wedge \neg\underline{\Diamond}^{\pm}\delta$$

The language of this expression is defined in section 2, and the expression itself is explained in full in section 3.6. For now it suffices to point out that the expression is a conjunction whose left conjunct algebraically describes a two-way channel and whose right logically describes the with-disconnect property. M^* is the set of all strings of messages, and $M^* \times A$ stamps each A-to-B message m with its origin A by converting m to (m, A). This sequence of stamped messages is then expanded to a one-way channel, the AB channel, via $(M^* \times A) \times AB$, a product. The $\|$ forms a two-way channel as the "sum" of the two concurrently executing one-way channels from A to B and from B to A respectively. The $\underline{\lambda}$ linearizes (serializes) each end of the channel, and the π makes the two-way channel abortable. The expression $\underline{\Diamond}^{\pm}\delta$ is the predicate "there was a strictly previous colocated disconnect" where δ is the predicate, assumed given, that expresses the property of an event being a disconnect message.

1.2 Why Partial Orders?

Strings arise naturally in modelling ongoing sequential computation, whether the symbols in the string correspond to states, commands, or messages. Thus the string uvu may model the sequential execution of three commands u, v, u, or a transition from state u to state v followed by a transition back to u, or a sequence of three messages u, v, u transmitted sequentially on some channel.

Strings are linearly ordered sets, or rather linearly ordered multisets (since repetitions are possible), of symbols from some alphabet. In unison with the workers mentioned at the end of this section we advocate partial orders in place of linear orders in modelling concurrent computation. At present however partial orders have nowhere near the popularity of linear orders for modelling concurrent computation. This could be for any of the following reasons.

(i) Languages and their associated operations, particularly union, concatenation, Kleene star, and shuffle, provide a natural model for the corresponding programming language control structures: choice, sequence, iteration, and concurrency. The behavior of languages under these operations has been studied intensively for more than two decades. Thus languages provide a familiar and well-understood model of computation. In this model the linear order on the elements of a string is interpreted as the linear temporal order of events, and the operations on languages may be interpreted as control structures: concatenation as begin-end sequencing, star as iteration, shuffle as concurrency, etc.

(ii) Every poset is representable as the set of its linearizations. This theorem would appear to confer on linear orders the same representational ability as partial orders.

(iii) Linear orders appear to be faithful to physical reality. In the practical engineering world, as opposed say to the physicist's relativistic world, instantaneous events have a well-defined temporal order, justifying the assumption of linearly ordered time. Furthermore, in any rigid system temporal order is well-defined even in a relativistic model. Any departures from rigidity are assumed to be sufficiently minor in practice as to justify adhering to a linear-order model.

Reason (i) would lose most of its force if partial orders were to be equipped with operations analogous to those of formal languages that could be interpreted as programming language control structures. This is just what this paper does; some of the operations on pomsets that we introduce correspond to more or less familiar programming language constructs, others are merely candidates for possible future programming or hardware languages.

Reason (ii) is based on the following well-known theorem, which shows that a partial order can be represented as the set of its linearizations.

Theorem 1. The intersection of the linearizations of a partial order is that partial order.

(For the purposes of defining intersection, a partial order is considered to be its graph, that is, the set of all pairs (a, b) such that $a \leq b$.)

This theorem is easily proved under the (non-obvious) assumption that every partial order has at least one linearization, by showing that any partial order in which a and b are incomparable can be extended to one in which $a < b$ and to another in which $b < a$.

This theorem about posets runs into two difficulties when trying to apply it to processes modelled as sets of pomsets. The theorem generalizes neither to *pomsets* nor to sets of *posets*, and *a fortiori* not to sets of pomsets. We will return to this issue in section 2.6, after the necessary definitions have been given.

Reason (iii), that the engineer's world is linear in time, fails in at least three situations: complex systems, nonatomic events, and relativistic systems. Beyond a certain scale of system complexity it becomes infeasible to keep thinking in terms of a global clock and a linear sequence of events. A cover story in the magazine Electronics[(3)] describes a growing trend in the design of logic circuits to eliminate global clocks and rely more on self-timed circuits. On a larger scale asynchrony has been with us for a long time. When a large number of computers communicate with each other over channels whose delay is several orders of magnitude greater than the clock time of each computer, the concept of global time provides neither a faithful account of the concurrent computation of all those computers nor even a particularly useful one. There is no reason to suppose that the various instructions streams of these computers are interleaved to form one stream. Indeed it is much more convenient, both conceptually and computationally (e.g. when computing with such streams as part of reasoning about them) just to lay down these streams side by side and call this juxtaposition of streams a model of their concurrent execution. Data flowing between the computers may augment the order implicit in the juxtaposition, but this relatively sparse augmentation of the order is motivated by the actual mechanics of communication, unlike the more stringent and totally artificial ordering requirement of completely interleaving the streams.

A concrete situation that may make this more compelling consists of a ship rolling somewhere in the Pacific, in satellite communication with another ship in the Indian Ocean. The events on the buses of the computers on each ship take place with a precision measured in nanoseconds, but the delay in getting a packet from one computer to another may be on the order of a second or more. The idea that the totality of events in the two computers has a well-defined linear ordering can have no practical status beyond that of a convenient mathematical fiction. Our position is that it is neither convenient nor mathematically useful. It is just as convenient, and more useful, to work with partial orders.

Nonatomic events provide another situation where linear orders break down. An event may be more complex than a moment in time. It may be an interval, in the sense of a convex subset of a linear order. It may be a set of intervals, such as a game punctuated by timeouts or a TV movie punctuated by commercials. More generally still it may be some arbitrary set of moments. However even for such complex events it still makes sense to say that one event may precede or follow another, meaning that every moment of the first event precedes every moment of the second. Yet such events are clearly not linearly ordered.

Relativity provides yet another situation where time is not linearly ordered. In any nonrigid system, that is, one whose components are moving with respect to each other, simultaneity ceases to be well-defined and two moving observers can report contradictory orders of occurrence of a pair of events. Any system nontrivially subject to relativistic effects is a candidate for a partially ordered model of computation. Of course many systems will not be so subject, but we see it as an advantage of the partial-order approach that it applies equally well to relativistic and Newtonian (global-time) situations.

In addition to our responses to (i)-(iii), we have the following additional reasons for preferring partial orders.

(iv) Some concepts are only definable for partial orders, in particular orthocurrence, which amounts to the direct product of pomsets, which we define in full later. The solution given above to the problem of specifying the two-way-channel-with-disconnect contains two essential uses of orthocurrence, along with two less essential uses. The concept is an extremely natural and useful one for partial orders, but it is not at all obvious how one would go about defining it in a linear-order model, or even whether it is definable.

(v) A serious difficulty with the interleaving model is that exactly what is interleaved depends on which events of a process one takes to be atomic. If processes P and Q consist of the single atomic events a and b respectively then their interleaving is $\{ab, ba\}$. However if the same events a and b are perceived by someone else not to be atomic, by virtue of having subevents, then P and Q have a richer interleaving than $ab \cup ba$. It is reasonable to consider instantaneous events as absolutely atomic, but we would like a theory of processes to be just as usable for events having duration or structure, where a single event can be atomic from one point of view and compound from another. In the partial-order model what it means for two events to be concurrent does not depend on the granularity of atomicity.

(vi) In some situations pomsets appear to be easier to reason about than strings. For example it is relatively straightforward to axiomatize the equational theory of pomsets under the operations of concurrence and concatenation (Theorem 5.2[4]). The corresponding theory for strings has resisted all attempts at its axiomatization. Gischer and the author have both worked extensively on the problem of whether this simply described theory has a finite axiomatization. The problem has been posed on two occasions at the (San Francisco) Bay Area Theory Symposium, generating interest but no answers in more than eighteen months.

We are not alone in our advocacy of partially ordered sets in one form or another for modelling concurrency. Partial orders make an early appearance in Greif's thesis[5]. C.A. Petri has advocated this view of computation for many years. The report by Best et al[6] and the book by Reisig[7] give excellent accounts of how partial orders may be used with Petri nets. Winskel's theory of event structures [8,9,10] concerns partial orders on events in Petri net models of computation. Pinter and Wolper consider partial orders as a model of temporal logic[11], with an interpretation of incomparability that is very close to ours. Lamport[12] makes several arguments similar to our own arguments for a nonlinear view of time in modelling communicating processes. Van Benthem[13] devotes an entire book to the comparison of the point and period views of time, making the point at the outset that there are other natural ways to think about temporal events than as instants. Whitrow[14] discusses many models of time, including "time as a type of serial order," considering both linear and partial orders. Mazurkiewicz in a recent paper[15] provides several references to work on this notion in modelling concurrency, dating back to 1977; Mazurkiewicz's own trace model comes closest to our pomset model, though it seems to us to lack the full generality of order obtainable with pomsets, e.g. Gischer's axiom $N(p,p,q,q) = pq \| pq$ does not hold for pomsets but it holds for Mazurkiewicz traces because these are derived as quotients of monoids, which do satisfy the axiom, and quotients preserve equations. The term used here, "partially ordered multiset," was introduced by us[16]; we used it to formalize Brock and Ackerman's[17] extension of Kahn's model[18,19], and subsequently extended this use[20,1]. Various equational theories of pomsets, sets of pomsets, ideals of pomsets, and their relation to theories of languages, were studied in depth by Gischer [4], whose thesis remains by far the most extensive treatment of the algebraic theory of pomsets. In particular he determines for each of a number of equational theories of pomsets and of languages, under concatenation, concurrence, and other pomset-definable operations, whether or not they are finitely axiomatizable.

1.3 The Concept of Pomset

A string may be regarded as a finite linearly ordered multiset. Multisets rather than sets are needed because a string may contain more than one occurrence of the same symbol. This view of strings has a natural generalization in which "linear" is replaced by "partial," yielding the notion of finite *partially* ordered multiset. Another simple generalization is to omit "finite." By analogy with the usual contraction of "partially ordered set" to "poset" we contract "partially ordered multiset" to "pomset." We call a set of pomsets a process, generalizing the notion of language as a set of strings.

As we shall see in more detail in the definitions below, a pomset may be defined to be a vertex-labelled partial order up to isomorphism. (Up-to-isomorphism says that the identities of the vertices are unimportant and they can be treated as anonymous points). For our application of pomsets to modelling concurrency, the interpretation we have in mind for an element of this partial order is an *event*. The interpretation of $e < f$ is that event e precedes event f. For structured events, e.g. events that are intervals rather than instants, this means that the whole of e must precede the whole of f, i.e. e must complete before f can begin. Two events incomparable under this order may then be interpreted as being permitted to occur concurrently.

1.4 Pomset Operations

The mental leap from strings to pomsets has much in common with that from reals to complex numbers. In particular one can encounter complex numbers without previously having seen a definition of the concept, via operations on real numbers, the canonical example being to apply square root to a negative number. A similar situation obtains with pomsets, as Figure 1 (a) illustrates in the application of "pomset multiplication" to two strings. In this figure vertices denote pomset events and the order on the events is given by the reflexive transitive closure of the relation implied by the arrows.

$$
\begin{array}{ccccccc}
0 & & T & & (0,T) & \rightarrow & (0,R) \\
\downarrow & \times & \downarrow & = & \downarrow & & \downarrow \\
1 & & R & & (1,T) & \rightarrow & (1,R)
\end{array}
\qquad\qquad
\begin{array}{ccccc}
0 & & T & & 0\ T \\
\downarrow & \| & \downarrow & = & \downarrow\ \downarrow \\
1 & & R & & 1\ R
\end{array}
$$

(a) $\qquad\qquad\qquad\qquad$ (b)

Figure 1. (a) Orthocurrence and (b) concurrence, of strings 01 and TR.

In Figure 1 (a) we have "multiplied" two strings 01 and TR to yield not a string but a poset. The multiplication, which we call orthocurrence, may be characterized as the direct product of partial orders. This multiplication has a very natural and useful interpretation: if we interpret TR as modelling the sequence Transmit-then-Receive, and 01 as the message 0 followed by the message 1, then the product is immediately recognizable as the four events Transmit 0, Receive 0, Transmit 1, Receive 1. Furthermore the order is equally recognizable as the necessary temporal order on these four events! We shall call pomset multiplication *orthocurrence*.

The need for pomsets rather than posets becomes apparent when we substitute the string 00 for 01 in this example. We then have four events consisting of two occurrences of each of the actions Transmit 0 and Receive 0. This constitutes a multiset with four elements but with only two distinct elements.

Because of this possibility of repetition of actions we draw a distinction between events and actions. Actions label events, that is, there is a labelling function from events to actions. An event is an instance or occurrence of its action.

Nonstrings may be produced from strings not only by multiplication but also by addition, as Figure 1(b) illustrates. We call pomset addition *concurrence.*

In this paper we introduce a number of other pomset operations besides concurrence and orthocurrence. They divide naturally into three main groups that we may call combinatorial, Boolean, and homomorphisms. In addition to these operations, which constitute the algebraic portion of a language for describing pomsets, we propose the use of first-order logic as the logical portion of our language. We give here a brief overview of all these constructs, leaving their formal definitions to the next section.

The number of operations is quite large compared to the typical logic of programs. However this seems to come with the territory; there are a number of constructs that make good sense for pomsets but that do not have a formal analog for the more commonly used linear models of computation, whereas almost any operation that makes sense for strings seems to generalize smoothly to pomsets, so the growth seems unavoidable.

Combinatorial operations. The combinatorial operations on pomsets manipulate vertices while keeping their labels more or less constant. Several of these operations come straight from formal language theory, namely concatenation $p;q$ or just pq, Kleene star p^*, prefix closure $\pi(p)$, and reverse p^-. These pomset operations when applied to finite linearly ordered multisets, i.e. strings, have their standard meaning. They are joined by several more pomset and process operations which do not correspond as closely, if at all, to string or language operations, namely concurrence (sum) $p\|q$, orthocurrence (product) $p\times q$, $N(p)$ (the quaternary N operation), dagger $p\dagger$ (concurrence closure), double-dagger $p\ddagger$ (expansion closure), augment closure $\alpha(p)$, linearizations $\lambda(p)$, reverse p^-, local linearizations $\underline{\lambda}(p)$, and local concatenation $p\underline{;}q$.

The concurrence $p\|q$ denotes the process consisting of two concurrently executing processes p and q, with no order constraints at all between their respective events. It plays the role in the pomset approach that shuffle or interleaving plays for languages, namely providing the basic concurrency operator. (Our $\|$ notation here agrees with that of Gischer[4] but not with that of a paper of ours[1] where we used $+$ instead; we have reverted to $\|$ to avoid confusion with the widespread use of $+$ to denote choice. We stick to $\cup$ for our choice operator however to emphasize that it is a pure union.)

The concatenation $p;q$ is the variant of concurrence in which every event of p must finish before any event of q may begin. This is the basic begin-end sequencing operator.

Prefix closure $\pi(p)$ makes a process abortable by permitting it to get only part of the way through its computation. Augment closure $\alpha(p)$ weakens the notion of concurrency of two events to mean absence of temporal constraint, so that one may precede the other. Augment closure is useful when concurrency of two events is to be interpreted as simultaneity, or at least as temporal overlap, meaning that neither may precede the other; under this interpretation one applies augment closure to a process in order to "turn off" that interpretation. Both closures are normally applied to processes that are to be interconnected to form a system. The linearization $\lambda(p)$ is the linear part of the augment closure, and for finite p is a language (set of strings).

Star means indefinite iteration while dagger means indefinite concurrence; in each case indefinitely (but finitely) many copies of the argument are "spawned," and they run sequentially or concurrently respectively. Double-dagger is a variant of dagger that is mostly used in the context $\Sigma\ddagger$, where it means all finite pomsets on alphabet Σ, analogously to Σ^* meaning all strings on Σ. Since for such processes of purely atomic behaviors we may express $\Sigma\ddagger$ as $\alpha(\Sigma\dagger)$ the utility of $\ddagger$ as a separate operation is not clear to us.

Orthocurrence is used to construct the kind of orthogonal concurrency associated with the lifetime of ports and messages. In a communication channel one can observe a port and see messages go by, or observe a message and see ports go by. These two behaviors go on in parallel, but the formal operation of concurrence does not describe how they combine to form the total channel behavior. The operation we use instead is the dual of concurrence, namely orthocurrence. Orthocurrence is to concurrence as direct product is to disjoint union.

The N operation is a four-way concurrence with three order constraints between the four processes in the shape of an N; it crops up in various unexpected places. The reverse operation runs a process backwards.

The two local operations rely on a notion of colocation of events, an equivalence relation on events which we take to be defined by a fixed equivalence relation on the alphabet described later. Local linearization $\underline{\lambda}(p)$ enforces serialization (interleaving) of colocated events of p while local concatenation $p\underline{;}q$ is a weaker variant of concurrence than concatenation in that the p-before-q requirement is applied only locally, namely to colocated pairs of events. Note that $p\underline{;}q$ cannot be expressed as either $f(p;q)$ or $f(p\|q)$ for any operation f because $p\|q$ need not contain the information as to which of p or q each event came from.

Boolean operations. Processes (sets of pomsets) may be combined using union $P \cup Q$, intersection $P \cap Q$, difference $P - Q$ or $P \wedge \neg Q$ for clarity, etc. These operations are exactly as for languages. Union means nondeterministic choice of a behavior from either argument, intersection yields a process that can only do what both arguments can do (a form of concurrency that we use later in defining the semantics of a system), and difference introduces a negation operator.

Homomorphisms. As with the four first-mentioned combinatorial operations and the Boolean operations, homomorphisms and their inverses exactly generalize the formal language notion of homomorphism. A pomset homomorphism carries pomsets to pomsets by expanding each vertex v of a given pomset to a pomset determined by $\mu(v)$. Thus if each vertex is expanded to a two-vertex pomset the net effect of the homomorphism is to double the number of vertices. A process homomorphism carries each pomset to a set of pomsets by expanding each vertex v to any one of a set of pomsets determined by $\mu(v)$. Pomset and process homomorphisms are exactly what are obtained by generalizing string and language homomorphisms to pomsets.

In this paper we do not contemplate any particular language constructs for specifying particular homomorphisms. As such they have the same status as atomic propositions in propositional calculus: we don't know how to specify them, only how to use them. Since one of the more important applications of homomorphisms is in specifying systems of communicating processes, a first step towards a language for homomorphisms would allow the specification of those homomorphisms defined by finite network topologies. Another important application is in defining functional processes; again a language for this is called for.

An inverse homomorphism is just the inverse of a homomorphism, in the usual sense of the inverse of a function. The inverse of a pomset homomorphism maps pomsets to processes (sets of pomsets). We use it only in the semantics of systems, defined as an intersection of inverse homomorphisms of processes.

Pomset Logic. The language of first order logic may be used to talk about a pomset by regarding the pomset as a structure (a set together with some relations). The individuals of the structure (the elements of the set) are the events (vertices) of the pomset. In one basic form the language consists of the relation $\leq$ and for each symbol $\sigma \in \Sigma$ the predicate $\sigma(u)$, defined as $\mu(u) = \sigma$. First order formulas are built up in the usual way from this language using variables ranging over events, quantifiers, and Boolean connectives.

Alternatively the temporal logic fragment of this language may be used, in which in place of quantifiers and event variables there is $\Diamond\phi$, denoting $\exists v[u \leq v \wedge \phi(v)]$, a predicate in the event variable u which holds for just those events followed by an event ϕ. This has the advantage of avoiding mentioning events explicitly; temporal logic is to the first order logic of events as combinatory logic is to the lambda calculus. In this case the basic language consists of unary predicate symbols, $\Diamond$, and Boolean connectives, the language of propositional temporal logic in which events are not referred to explicitly.

This language, whether full first order logic or the temporal logic fragment, can express predicates over events. Such a predicate may be interpreted as a predicate over pomsets by defining it to hold of a pomset just when it holds of all events of that pomset.

1.5 Hidden Nondeterminacy and Fairness.

Implicit in our two-layered notion of a process as a set of pomsets of events is a two-sided distributivity axiom, that order may be distributed over choice, i.e. concatenation over union. This axiom is appropriate for partial correctness but not always for issues relating to termination and deadlock. Milner[21] and others have argued at length the need for nondistributivity, namely $a(b \cup c) \neq ab \cup ac$. The distinction being drawn between these two sides has to do with the timing of the decision of which side of the union to take; the left side decides after the a and the right before. By deciding later the left side can only be better informed about the choice, and may thereby be able to foresee and sidestep a deadlock. At this level of abstraction the decision appears to be made nondeterministically. Further, if the decision happens early (right side) the associated choice is "hidden" in the sense that the a action does not reveal the outcome of the choice, or even that a choice has been made. This situation may be called *hidden nondeterminacy.*

Modelling hidden nondeterminacy appears to entail a significant degree of conceptual complexity. For now we wish to work out the other aspects of the pomset approach without the distraction of this complication, leaving its treatment to later.

This is not however to say that our model will be unusable until nondistributivity is catered for. Nondistributivity appears to be needed only to draw distinctions involving hidden nondeterminacy. While a language for *describing* a system implementation will presumably need to describe hidden nondeterminacy, this is because it will need to describe system pathologies in general, of which hidden nondeterminacy is one. However a language for *specifying* a system need not make provision for explicitly discussing pathologies since they may be proscribed globally. The purpose of a language for specifying an object is to distinguish between the elements of the set of *acceptable* objects. The corresponding situation for restaurants is that the word "poison" need not be included in the vocabulary of menus since there is a global understanding between restaurateurs and their clients that poison is not one of the options. On the other hand the restaurant kitchen's first-aid book should cover the case of a poisoned customer. Menus are to first-aid books as specification languages are to system description languages; only the latter need be able to express pathologies such as hidden nondeterminacy. For now we are interested primarily in applying our language to system specification, though we would like to extend it to treat pathologies related to nondistributivity when we see a straightforward way.

In addition to hidden nondeterminacy there is the issue of fairness, which is getting to be a *sine qua non* of any concurrency model, at least among theoreticians. We have two reasons for not considering fairness here. The first, applicable only to specification languages, is the "poison" argument above; we want all our systems to be fair and so don't need the concept in a specification language. The second, applicable to more general system description languages, is that the impact of fairness on the participants in a computation is felt only after an infinite time. Unfairness is only possible with infinite behaviors whereas we can only experience finite behaviors. Of course we do not want to participate in even an initial segment of an unfair infinite computation, but merely replacing such a computation by a fair one provides no guarantee at all that we will be any happier. What is needed instead is some notion of local or bounded or rapidly converging fairness. Pending the development of some algebraically and/or logically tractable notion of fairness that is more useful than fairness-at-infinity, we have not attempted to deal with fairness in our model.

1.6 Inappropriateness of Operational Semantics

Our approach to modelling concurrency is *denotational*, or extensional, in the sense that we have a concrete mathematical model of a computational behavior, along with operations on behaviors that yields a particular algebra of behaviors. An alternative approach has been advocated by Milner[21]. In Milner's "operational" approach the meaning of expressions is defined by reductions between expressions, analogous to β-reduction for the λ-calculus. This semantics is operational in the sense that each reduction step may be interpreted as a step of a global interpreter of the given concurrent program.

This model implicitly forces an interleaving view of concurrent computation. The global computation is serialized by the choice of order of reductions. Our objection to this model is therefore the same as for any interleaving model. In a computation carried out by two fast computers miles apart, Milner's reductions do not correspond to what actually takes place in the system as a whole. In reality there is no global interpreter carrying out those steps; instead computation steps are performed locally and mostly independently. Our complaint applies to any operational semantics that is similar to Milner's in being defined to act on the whole program.

§2 Definitions

2.1 Pomsets

The following definition of pomset is due to Gischer[(4)].

A *labelled partial order* (lpo) is a 4-tuple $(V, \Sigma, \leq, \mu)$ consisting of

(i) a *vertex set* V, typically modelling *events*;

(ii) an *alphabet* Σ (for symbol set), typically modelling *actions* such as the arrival of integer 3 at port Q, the transition of pin 13 of IC-7 to 4.5 volts, or the disappearance of the 14.3 MHz component of a signal;

(iii) a *partial order* $\leq$ on V, with $e \leq f$ typically being interpreted as event e necessarily preceding event f in time; and

(iv) a *labelling function* $\mu : V \to \Sigma$ assigning symbols to vertices, each labelled event representing an *occurrence* of the action labelling it, with the same action possibly having multiple occurrences, that is, μ need not be injective.

A *pomset* (partially ordered multiset) is then the isomorphism class of an lpo, denoted $[V, \Sigma, \leq, \mu]$. By taking lpo's up to isomorphism we confer on pomsets a degree of abstractness equivalent to that enjoyed by strings (regarded as finite linearly ordered labelled sets up to isomorphism), ordinals (regarded as well-ordered sets up to isomorphism), and cardinals (regarded as sets up to isomorphism).

2.2 Types of Pomsets

It will be convenient to regard all of the following structures as kinds of pomsets. This provides some useful "coercions;" for example we may construct a power set and then interpret its elements not only as sets but as pomsets.

Multiset	Pomset with a minimal (i.e. empty) order
Tomset	Pomset with a maximal (i.e. total) order
String	Finite tomset
Poset	Pomset with an injective labelling
Set	Poset that is also a multiset
Atom	A singleton pomset (both a set and a string)
Unit	The empty pomset ϵ (hence the empty string, and empty set)

2.3 Processes

A *process* is a set of pomsets, just as a *language* is a set of strings, and an *n-ary relation* is a set of n-tuples. In each case the set structure can be regarded as modelling variety of one or another kind of behavior.

We write $\Sigma\ddagger$ for the set of all finite pomsets with alphabet Σ, by analogy with Σ^* for the set of all strings with alphabet Σ. The set of atoms of $\Sigma\ddagger$ is just Σ.

Set-of-atoms ambiguity. There is an ambiguity when forming a set of atoms as to whether the set is meant to be a single pomset or a process consisting of a set of atomic pomsets. Under the former interpretation the concatenation $\{0,1\}\{2,3\}$ is a single pomset with four events, under the latter it is the process $\{02,03,12,13\}$. When the ambiguity arises syntactically we resolve it by always collecting pomset elements using $\|$, writing $0\|1$ rather than $\{0,1\}$. When the ambiguity arises in giving semantics, as in the section on systems, where the inverse of a translation between alphabets is considered a function mapping atoms to pomsets rather than a function mapping atoms to processes, we say explicitly which interpretation is meant.

The set-of-atoms ambiguity does not arise in formal language theory because an unordered set of atoms cannot be mistaken for a string.

When a pomset is used in a context that appears to require a process, the pomset is interpreted as a process consisting of a single pomset. Thus $\{0,1\}2$ denotes $\{0,1\}\{2\} = \{02,12\}$. This convention is as in formal language theory.

2.4 Operations of Pomset Algebra

We give here the formal definitions of those operations on pomsets and processes that we have found useful and/or interesting. We collect them here under the rubric of pomset algebra not so much because we have a fixed class of algebras in mind but rather to draw a distinction with the constructs of pomset logic described in the next section. Informal descriptions and uses of the operations are listed in section 1.4.

All the combinatorial operations in the following are defined to map pomsets to either pomsets or processes. Just as with languages concatenation of strings generalizes to concatenation of languages, so do operations on pomsets generalize to operations on processes. For example the orthocurrence $P \times P'$ of processes P and P' is $\{p \times p' | p \in P, p' \in P'\}$. With this in mind we will always define the action of operations on single pomsets, but will sometimes give examples of their action on processes.

Concurrence. The concurrence $p\|p'$ of two pomsets is defined as $[V,\Sigma,\leq,\mu]\|[V',\Sigma',\leq',\mu'] = [V\cup V',\Sigma\cup\Sigma',\leq\cup\leq',\mu\cup\mu']$ where V are V' are assumed to be disjoint. This assumption entails no loss of generality since pomsets are only defined up to isomorphism. It follows that the cardinality of a concurrence is the sum of the cardinalities of its constituent pomsets, that $\mu\cup\mu'$, as the union of the graphs of the labelling functions μ and μ', is a function, and that $\leq\cup\leq'$ is a partial order (no interference between p and q and so no opportunity to violate transitivity).

Concatenation. The concatenation $p;p'$, or just pp', is as for concurrence except that instead of $\leq \cup \leq'$ the partial order is taken to be $\leq \cup \leq' \cup(V \times V')$. This forces every event of p to precede every event of q. This larger order is still a partial order.

Syntactically concatenation has higher precedence than any other binary operation (so $p\|qr = p\|(qr)$) and lower than any unary operation (so $pq^* = p(q^*)$). For all other potentially ambiguous combinations we will use parentheses for disambiguation in preference to having an elaborate hierarchy of syntactic precedences. The ubiquity of concatenation justifies its special treatment.

$$
\begin{array}{c} 0 \\ \downarrow \\ 1 \end{array} \; ; \; \begin{array}{c} T \\ \downarrow \\ R \end{array} \; = \; \begin{array}{c} 0 \\ \downarrow \\ 1 \\ \downarrow \\ T \\ \downarrow \\ R \end{array}
\qquad\qquad
\begin{array}{c} 0 \\ \\ 1 \end{array} \; ; \; \begin{array}{c} T \\ \\ R \end{array} \; = \;
\begin{tikzcd} 0 \arrow[r] \arrow[dr] & T \\ 1 \arrow[r] \arrow[ur] & R \end{tikzcd}
$$

Figure 2. Examples of Concatenation

The N operation. The N of four pomsets $N(p,q,r,s)$ is $pr\|qs$ with order augmented to make all events from p precede all events from s. Figure 3 illustrates $N(a,b,c,d)$ for atoms a,b,c,d, an N-shaped pomset. It is the smallest pomset not expressible in terms of its atoms using concurrence and concatenation.

$$
\begin{tikzcd} a \arrow[d] \arrow[dr] & b \arrow[d] \\ c & d \end{tikzcd}
$$

Figure 3. The pomset $N(a,b,c,d)$.

An identity involving N is $\lambda(N(p,p,q,q)) = \lambda(pq\|pq)$ [(4)]. Noting that $N(p,p,q,q) \neq pq\|pq$, this observation of Gischer answers in the negative the interesting question as to whether theoremhood of $\lambda(p) = \lambda(q)$ implies theoremhood of $p = q$ for all pomset expressions p and q built up from pomset variables using the language introduced here. This is discussed further in section 2.6.

Dagger. $p\dagger$ is not a pomset but a process (set of pomsets), namely the set of all pomsets expressible as a concurrence of finitely many copies of p. That is, $p\dagger = \{\epsilon, p, p\|p, p\|p\|p, \ldots\}$.

Example: The dagger of the process consisting of the atomic pomsets 0 and 1 is given by $\{0,1\}\dagger = \{\epsilon, 0, 1, 0\|0, 0\|1, 1\|1, \ldots\}$.

Star. p^* is Kleene star: $p^* = \{\epsilon, p, pp, ppp, \ldots\}$. It is to $p\dagger$ as concatenation is to concurrence.

Example: $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \ldots\}$.

Orthocurrence. The orthocurrence $p \times p'$ is defined as $[V, \Sigma, \leq, \mu] \times [V', \Sigma', \leq', \mu'] = [V \times V', \Sigma \times \Sigma', \leq \times \leq', \mu \times \mu']$. Analogously to the action of $\cup$ on $\leq$ and μ in the definition of concurrence, namely the union of their graphs, the action of $\times$ on $\leq$ and μ is the Cartesian product of their graphs. That is, $(u, u') \leq (v, v')$ in $p \times p'$ just when $u \leq v$ in p and $u' \leq' v'$ in p', and $\mu(u, u')$ in $p \times p'$ is $(\mu(u), \mu'(u'))$. Orthocurrence is to concurrence as Cartesian product is to disjoint union.

Prefix closure. As with p^* and $p\dagger$, $\pi(p)$ is not a pomset but a process, consisting of the set of prefixes of p. q is a prefix of p, written $q \leq_\pi p$, when q is obtainable from p by deleting a subset of the events of p, provided that if event u is deleted and $u < v$ then v is also deleted.

Example: $\pi(0\|12) = \{0\|12, 12, 0\|1, 0, 1, \epsilon\}$

Augment closure. $\alpha(p)$ is the set of augments of p. q is an augment of p, written $p \leq_\alpha q$, when q differs from p only in its partial order, which must be a superset of that of p.

Example: $\alpha(0\|12) = \{0\|12, 012, 102, 120, 1(0\|2), (0\|1)2\}$.

Example identity: $p\|q \leq_\alpha pq$.

For the most part we have been avoiding theorems and proofs in this paper. Nevertheless the following theorem and proof is indicative of the methods used for proving things about pomsets.

Theorem 2. $\alpha(\pi(p)) = \pi(\alpha(p))$.

Proof. Suppose $q \in \alpha(q')$ where $q' \in \pi(p)$. Construct $q'' \in \alpha(p)$ by adding to p just those edges that α contributed to q' and taking transitive closure, possible since p has all the vertices q' does. Now delete the same vertices from q'' that were deleted from p to yield q', possible since all edges added in forming q'' must precede those vertices. Furthermore none of those added edges will be deleted here. Since we have added the same edges and deleted the same vertices the result must be q, showing that $\alpha(\pi(p)) \subseteq \pi(\alpha(p))$.

Conversely suppose $q \in \pi(q')$ where $q' \in \alpha(p)$. Construct $q'' \in \pi(p)$ by deleting the same vertices from p that were deleted from q' to yield q, possible since p is less constrained than q' for the purposes of taking prefixes. Now augment q'' with just those edges e that were added to p to form q' provided both ends of e are in q''. The result has the same vertices as q, and the added edges will be just those that that were added to q' and that remained in q after the deletions, showing that $\pi(\alpha(p)) \subseteq \alpha(\pi(p))$. []

Linearizations. The set $\lambda(p)$ of linearizations of p is the set of all linear augments of p.

Example: $\lambda(0\|12) = \{012, 102, 120\}$.

Reverse. The reverse p^- is obtained from p by replacing $\leq$ by its converse.

Example: $(0\|12)^- = 0\|21$.

Colocation. The next two operations are only defined for alphabets of the form $\Sigma = D \times C$. The intended interpretation is that C be a set of locations. We shall say that two events are *colocated* when each of their labels is a pair and the two labels agree in their second component.

Local linearization. Sometimes it is appropriate for a concurrence to have some interleaving, typically of colocated events. We say that an augment q of p is *locally linear* when all colocated pairs of events u, v in q are linearly ordered in q. The *local linearizations* of p are the *minimal* locally linear augments of p, that is, those which are not augments of any other locally linear augments of p.

Example. Abbreviating $(0,a), (1,b), (2,a), (3,b)$ to 0,1,2,3 respectively, $\underline{\lambda}(0\|1\|23) = \{N(2,1,0,3), 2(0\|31), 0231, (02\|1)3\}$.

Local concatenation. Sometimes it is appropriate for concatenation to enforce order only locally. The local concatenation $p \underline{;} p'$ of p and p' is that augment of $p\|p'$ obtained by adding to the order all colocated pairs $(u, u') \in V \times V'$ and then taking the transitive closure of the result. (Note that this last step is unnecessary for ordinary or nonlocal concatenation.)

Example. With the abbreviations of the preceding example, $(0\|1)\underline{;}(23) = (02\|1)3$.

Remark: $p\underline{;}q \in \underline{\lambda}(p\|q)$.

Boolean. Processes (sets of pomsets) may be combined using union $P \cup Q$, intersection $P \cap Q$, difference $P - Q$, etc. These operations are exactly as for languages.

The next construct, pomset homomorphism, exactly generalizes the notion of string homomorphism. Following Gischer[(4)], we shall find it convenient to derive the notion of homomorphism from the notion of *expansion* of a pomset $p = [V, \Sigma, \leq, \mu]$ on alphabet Σ *through* a Σ-tuple t of pomsets all on a common alphabet Σ' (that is, t is a function mapping elements of Σ to pomsets on Σ'). This expansion yields a pomset on Σ'.

Informally, expansion is the result of substituting $t(\mu(u))$ for each u in p and flattening the resulting set of sets of events down to a single set of events, preserving as much of the order as possible. Formally, for each u in V, let v_u denote element v of $V_{t(\mu(u))}$ (an event of the pomset substituted for u). (Without loss of generality we may assume the $V_{t(\mu(u))}$'s are all pairwise disjoint.) Take W to be the set of all such v_u's for u ranging over V. Define $\leq_W$ on W such that $v_u \leq_W v'_{u'}$ just when either $u < u'$ in p or ($u = u'$ in p and $v \leq v'$ in $V_{t(\mu(u))}$). Define $\mu_W(v_u) = \mu_{t(\mu(u))}(v)$. Then the expansion of p through t is the pomset $[W, \Sigma', \leq_W, \mu_W]$.

Another way to define expansion is to form $\|_{u \in V} t(\mu(u))$, the concurrence of $t(\mu(u))$ over V, then to augment its order with $\bigcup_{u<v} V_{t(\mu(u))} \times V_{t(\mu(v))}$.

Example. The expansion of the pomset $0\|01$ through the pair $(ab, a\|b)$ (i.e. the function mapping 0 to ab and 1 to $a\|b$) is the pomset $ab\|ab(a\|b)$, and through the pair $(a\|b, ab)$ it is $a\|b\|(a\|b)ab$.

Homomorphisms. A *pomset homomorphism* from pomsets on Σ to pomsets on Σ' is determined by a fixed Σ-tuple t of pomsets, and is the function mapping p to the expansion of p through t. A *size-preserving* homomorphism is the special case in which t is a function from Σ to Σ'; it merely relabels the pomset, without having to change the vertex set. A *process* homomorphism is one for which t maps symbols to sets of pomsets; it carries each pomset p to the set of pomsets obtainable by expanding each vertex v in p to some pomset of $t(\mu(v))$ (two vertices with the same label need not be expanded to the same pomset). This in turn extends to a function from processes in the usual way: the process P is mapped to the union of those processes mapped to from some pomset of P. When all the pomsets are strings, pomset homomorphisms are just string homomorphisms and process homomorphisms are just language homomorphisms.

An inverse pomset homomorphism h maps a pomset p to the set of pomsets $\{q|h(q) = p\}$, and by the obvious extension, maps a set of pomsets to a set of pomsets. That is, inverse homomorphisms map processes to processes.

The pomset-definable operations. Homomorphisms are obtained from expansions by fixing the tuple argument t. If instead the pomset argument p is fixed, expansion becomes what Gischer[4] calls a *pomset-definable* operation of arity Σ on pomsets over Σ'. By taking Σ to be $\{0, 1\}$ we get all the binary pomset-definable operations. Among these are those defined by the pomsets $0\|1$ and 01, recognizable as respectively concurrence and concatenation. Likewise $N(0, 1, 2, 3)$, where $\Sigma = \{0, 1, 2, 3\}$, defines the operation N. Gischer[4] shows that the operations definable by finite pomsets has no finite basis. Of the operations we have defined previously, the only pomset-definable ones are $\|$, ;, and N.

Double-dagger. With the notion of pomset-definable operation we are now able to define $P‡$ for a process P. $P‡$ is the closure of P under all the pomset-definable operations. This is analogous to the closure of P under the single pomset-definable operation of concatenation, namely P^*.

Example: $\{0, 1\}‡ = \{\epsilon, 0, 1, 0\|0, 0\|1, 1\|1, 00, 01, 10, 11, 0(0\|1), N(0, 0, 1, 0), \ldots\}$.

Identities. (i) $p† \leq_\alpha p‡ \leq_\alpha \alpha(p†)$. (ii) $P^* \subseteq P‡$. (iii) $P† \subseteq P‡ \subseteq \alpha(P†)$. (iv) If Σ is a process consisting solely of atoms, $\Sigma‡ = \alpha(\Sigma†)$. (v) $\lambda(P†) = \lambda(P‡)$.

2.5 Constructs of Pomset Logic

First-order logic provides an alternative to pomset algebra for describing pomsets. As a simple example the algebraic expression $\Sigma^* a \Sigma^* b \Sigma^*$, where $a, b \in \Sigma$, may be expressed logically as $\forall uv[u \leq v \vee v \leq u] \wedge \exists uv[u \leq v \wedge a(u) \wedge b(v)]$. This asserts that in each pomset the order is total, and that there is an occurrence of symbol a preceding an occurrence of symbol b. Provided we apply the logical formula only to finite pomsets it expresses the same set of pomsets as the algebraic expression. In this case the algebraic expression is shorter than the logical, but it need not always be that way, as we shall see.

The logical language consists of formulas including atomic formulas consisting of predicate symbols applied to variables, and is closed under Boolean operations and quantification. A sentence (closed formula, one with no free variables) may be interpreted as a predicate over pomsets (e.g. the formula of the previous paragraph), in which case it may be interpreted as denoting the set of those pomsets it holds for, which is a process. In this way sentences may be used to name processes.

The meaning of such first-order formulas is defined in the standard way for first-order logic. Each pomset is treated as a structure. Variables, whether quantified or free, range over events; hence an n-ary predicate (one having n free variables) applies to n-tuples of events of a pomset. A 0-ary predicate (no free variables, i.e. closed) is a first-order sentence holding or not holding for the pomset as a whole. Where an open (nonzeroary) predicate is used in a context demanding a sentence (e.g. where the formula is intended to denote a process), the predicate is implicitly closed with a universal quantifier. In an expression such as $P \wedge \neg\phi$ where P is an expression of type process and ϕ is a formula with a free variable, there is a nontrivial ambiguity as to whether to coerce ϕ or $\neg\phi$, since it makes a difference whether the quantifier used to close the formula is inside or outside the negation. The rule is to keep building up formulas till the coercion is forced, in this case putting the quantifier outside the negation.

Temporal Logic. As an abbreviation for certain first-order formulas, there is a predicate "transformer" or modality $\Diamond\phi$ (sometimes written $F\phi$) mapping the unary predicate ϕ to the unary predicate ψ defined as $\psi(u) = \exists v[u \leq v \wedge \phi(v)]$. It means of u that some future event (future with respect to u) will satisfy ϕ. As usual $[]\phi$ abbreviates $\neg\Diamond\neg\phi$ and says of event u that $\phi(v)$ holds for all events $v > u$. These operators have been used to good effect by Pneuli[22,23] and many others in the modelling of concurrency. Originally it was conceived of solely for linear orders, but it has also been applied more recently to other orders.

We may apply various decorations to temporal modalities; all denote modifications to the order relation in the definition of the operator. $\Diamond^-$ indicates that the converse of $\leq$, should be used (temporal reversal). $\Diamond^+$ denotes that $<$ should be used in place of $\leq$ (strict future); with converse it becomes $\Diamond^{\pm}$. $\underline{\Diamond}$ denotes local future: $\underline{\Diamond}\phi(u)$ holds just when $\phi(v)$ holds for some future v colocated with u, which is to say that the order defining $\underline{\Diamond}$ is the intersection of $\leq$ with the equivalence relation of colocation.

The rule for closing open formulas with universal quantifiers carries over unchanged to temporal logic; thus $\Diamond\phi$, interpreted as a predicate over pomsets rather than events, asserts that every event in the pomset is followed later by some other event satisfying ϕ.

These various temporal operators provide convenient abbreviations for first-order predicates on pomsets. In addition they may often provide fragments of theories of such predicates for which the membership problem is computationally tractable.

Customarily, temporal logic is conducted in some fixed set of pomsets: typically Σ^ω (linear time logic) or the set of infinite trees on S (branching time logic). Temporal formulas are used to specify a subset of these, that is, a process. Processes can be combined and operated on only to the extent that formulas of temporal logic can be so combined (namely with Boolean connectives) and operated on (namely with temporal operators).

Our methods may be regarded as a form of temporal logic in which processes may be conveniently viewed as objects, where multiple processes may be discussed in the one sentence, where there exist many other combinations of processes besides Boolean, and where processes may combine hierarchically to permit naturally structured compositions of processes.

Conversely, from the perspective of a practicing temporal logician, pomsets may be regarded as providing an alternative class of models for temporal logic. The "branching" structure of pomsets should not however be confused with that of branching-time temporal logic; the former is conjunctive (all paths are taken) while the latter is disjunctive (only one path is taken).

2.6 Representability of Pomsets by Languages

In section 1.2 we promised to return to the possibility of representing processes using languages. There are two issues here. Can sets of posets be represented by languages, and can pomsets be so represented?

We know of no satisfactory method of coding a set of posets as a set of strings, efficiently or otherwise, in a way that preserves algebraic properties such as concatenation. (Notice that the coding given by Theorem 1 does have this property, whence the power of this theorem.) We remark however that there are more than $2^{(n/2)^2}$ posets on n elements, there being that many just by forming bipartite graphs with $n/2$ vertices on each side. Since there are only $n^n = 2^{n \log_2 n}$ strings of length n on an alphabet of n letters, the obvious counting argument suggests that strings of length closer to n^2 would be needed in a typical language coding an arbitrary set of n posets. With this in mind, simply writing out each partial order as the n^2 elements of an $n \times n$ Boolean matrix would seem to be close to best possible with respect to economy. The result of course has no useful algebraic properties whatsoever as far as language operations go; for example concatenating such strings does not yield the result of concatenating the posets. It is an interesting question as to what is possible here. In the absence of a solution to this we are unconvinced by arguments defending linear orders on the ground that they can encode partial orders, in the case of sets of partial orders.

The other issue is whether pomsets can be represented by languages. One simple answer is that the pomsets $a \| a$ and aa have indistinguishable linearizations. Hence the theorem about encoding posets as sets of linear orders does not generalize to pomsets in the obvious way.

Actually if a were atomic and $a \| a$ meant that the two copies of a could be performed in either order then $a \| a$ and aa really are equivalent. However one advertised feature of the pomset approach is that it does not depend on whether a behavior is regarded as atomic. If a could be seen to have internal structure then the two expressions would be different.

One quite general way to formalize this is to regard each symbol of a pomset as standing for a language. Then aa denotes the concatenation of the two languages while

$a \| a$ denotes their shuffle. More generally, if a pomset has n vertices then it denotes the shuffle of n languages, one corresponding to each label, with the shuffle restricted so that a string coming from the language on vertex u must precede one from v just when $u < v$ in the order.

Under this interpretation Gischer(4) has shown that pomsets are equivalent if and only if they are equivalent under the weaker interpretation in which only languages whose strings are of length 2 are considered. This captures the intuition that when interpreting pomset events as strings, only the start and the end of the string are needed to distinguish pomsets under this interpretation. Moreover he has shown that $N(a, a, b, b) = ab \| ab$ under this interpretation. Since these are distinct pomsets it follows that pomsets cannot be represented as languages in this way. We know of no other acceptable method of coding general pomsets as languages.

For the special case where all vertices with the same label are linearly ordered, Theorem 1 remains true, which we leave as an easy exercise. While this still leaves open the problem of coding sets of pomsets, this observation may be of some use.

§3 Applications.

In this section we show how the language is used to specify some simple processes.

3.1 Channels

A channel is a process in which events happen in pairs consisting of a transmission and a receipt of a datum d, expressed by labelling the two events with (d, A) and (d, B) respectively where A is the transmission end of the channel and B the receipt end. The pair is ordered $(d, A) < (d, B)$.

The channel behavior corresponding to a string $\sigma \in D^n$ of n data being sent down the channel is represented as $\sigma \times AB$, a pomset with $2n$ events. Figure 4 shows the behavior for the string $\sigma = 01101$.

$$\begin{array}{ccccccccc}
(0,A) & \rightarrow & (1,A) & \rightarrow & (1,A) & \rightarrow & (0,A) & \rightarrow & (1,A) \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
(0,B) & \rightarrow & (1,B) & \rightarrow & (1,B) & \rightarrow & (0,B) & \rightarrow & (1,B)
\end{array}$$

Figure 4. The channel behavior $01101 \times AB$.

If the language L consists of all strings that may be transmitted on a channel then $L \times AB$ denotes the process corresponding to that channel.

3.2 The Cons and CarCdr Processes

The Cons process has alphabet $D \times \{A, B, O\}$ where D is a set of data values and A, B, O are the three ports, respectively two input ports and an output port. Informally Cons behaves as follows. The first value output from O is the first value input from A; thereafter all outputs on O are taken in order from B. Further inputs on A are not accepted and therefore do not appear at all in the behaviors of Cons. Note that this is asynchronous Cons: many B inputs may have arrived and have to be buffered by $Cons$ before the A input arrives.

The typical behavior of this process can be diagrammed as in Figure 5.

$$
\begin{array}{ccccccc}
(3, A) & & & & & & \\
\downarrow & & & & & & \\
(3, O) & \rightarrow & (7, O) & \rightarrow & (2, O) & \rightarrow & (5, O) \\
 & & \uparrow & & \uparrow & & \uparrow \\
 & & (7, B) & \rightarrow & (2, B) & \rightarrow & (5, B)
\end{array}
$$

Figure 5. Typical Behavior of the Cons Process

We may represent this process as a sum of products (concurrence of orthocurrences). There are two products, corresponding to two channels from A to O and from B to O respectively. The AO channel passes only one datum and hence has the form $D \times AO$, the BO channel is an ordinary order-preserving channel and so is $D^* \times BO$. The sum is constrained so as to force the one O output from A (and hence all of the AO channel) to precede all the O outputs from B; thus it is a local concatenation. This yields:

$$Cons = (D \times AO)\underline{;}(D^* \times BO)$$

.

CarCdr. The dual of Cons is CarCdr, a process with one input I and two outputs A and B, which feeds its first input to A and the rest to B. Like Cons, CarCdr is a sum of products. We may easily obtain CarCdr from Cons as:

$$CarCdr = (D \times IA)\underline{;}(D^* \times IB)$$

.

The duality of Cons and CarCdr should be clear from the formulas.

3.3 The Merge and Spray Processes

Merge. A somewhat different example is given by Merge, which has inputs A and B and output O and merges its two stream inputs arbitrarily. Similarly to Cons, Merge is a sum of two products, in this case $D^* \times AO$ and $D^* \times BO$. Locally linearizing this sum has the effect of merging the outputs, the inputs already being locally linear. Hence we may define Merge as

$$Merge = \underline{\lambda}((D^* \times AO)\|(D^* \times BO))$$

Spray. We might call the dual of Merge Spray. Spray has input I and outputs A and B, and sends each of its inputs to an arbitrarily chosen output. It is given by

$$Spray = \underline{\lambda}((D^* \times IA)\|(D^* \times IB))$$

or more simply by Merge$^-$. (Why is CarCdr not expressible as Cons$^-$?)

3.4 Functional Processes

A functional process F which computes the function f repeatedly has one input port I and one output port O, and for each input datum x on I outputs $f(x)$ on O. It may be defined as

$$F = h(D^* \times IO)$$

where h is a pomset homomorphism which replaces each label (x, O) with $(f(x), O)$ and leaves the rest of the pomset unchanged. This process is a paradigm for all "memory-less" processes that merely compute some function of their single input.

This approach generalizes straighforwardly to an n-input process F computing the n-ary operation f repeatedly. F may be defined as

$$F = h((D^n)^* \times (I_0\|I_1\| \ldots \|I_{n-1})O)$$

where $h((x_0, \ldots, x_{n-1}), I_j) = (x_j, I_j)$ and $h((x_0, \ldots, x_{n-1}), O) = f(x_0, \ldots, x_{n-1})$.

3.5 Combinational Dataflow and Pipeline Semantics

In its most general form a net connects arbitrary processes using an arbitrary graph that may have cycles. The next section solves the problem of assigning a semantics to such a net in its full generality. Here we give a simpler treatment of the special case where the net is acyclic and each component is an n-ary functional process. This special case corresponds to the behavior of a so-called combinational circuit, one that when presented with an m-tuple of values at its m inputs eventually yields an n-tuple of values at its n outputs.

A naive way to specify what the circuit computes is to hold the inputs fixed and wait for the circuit to quiesce, at which moment the outputs are deemed valid. Once the outputs have been determined a new set of inputs may be presented, allowing the circuit to perform a new computation. In practice however, once an output of a component has been consumed that component can be freed up for the next computation. In this way the throughput of a complex circuit may be made significantly greater than just the reciprocal of its delay. This is the principle of pipelining.

The behavior of the pipelined case can be described as the homomorphic image of an orthocurrence of a sequence of m-tuples (constituting the inputs) with the dependency relation of the circuit. The dependency relation R_C for a circuit C having a set T of nodes is a poset with event set T ordered so that $s < t$ just when there is a path in the circuit from node s to node t (where paths travel through components from input to output). (For a poset it suffices to give V and $\leq$ since Σ may be assumed to be the vertex set V and μ the identity function on V.) The process is thus:

$$F = h((D^m)^* \times R_C)$$

where $h((x_0, \ldots, x_{m-1}), t) = (v_t, t)$ where v_t is the value eventually appearing at node t when the circuit is presented with $(x_0, \ldots, x_{m-1})$ and allowed to quiesce. Even though h is defined using quiescence semantics, the process given by this expression imposes only the minimal order constraint imposed by causality.

3.6 Two-Way Channel with Disconnect

We take for our concluding example the two-way channel with disconnect described at the beginning of the paper. In formalizing this problem we assume we are given symbols A and B denoting the two ends of the channel, the set M of all messages that may be transmitted on the channel, and a predicate δ on events that holds just for those events that are disconnect messages. Using these four objects and our language the desired channel may be expressed as

$$\pi \underline{\lambda}(((M^* \times A) \times AB) \| ((M^* \times B) \times BA)) \wedge \neg \underline{\diamondsuit}^{\pm} \delta$$

We may understand this expression bottom-up. Let E denote the set $\{A, B\}$ of ends. M^* is the set of all sequences of messages. $M^* \times o$ for $o \in E$ is the set of all sequences of origin-stamped messages, each of the form (m, o) consisting of the message m itself together with its origin o. (Orthocurrence of p with an atom is a somewhat degenerate application of orthocurrence in that all it really does is "stamp" each event of p with that atom.) $(M^* \times o) \times od$ is a channel of the kind we have already seen many of, with origin o and destination d. At this point the alphabet has become $(M \times E) \times E$, and stays that way for the remainder of the development. The action $((m, o), e)$ consists of the stamped message (m, o) being transmitted or received at e depending respectively on whether or not $e = o$.

The concurrence of the AB and BA channels expresses the two-way channel as the concurrent execution of the two one-way channels. We then linearize the events at each end with $\underline{\lambda}$, in order to have a sensible notion of time at each end for the purpose of defining disconnection of each end. Then we take the prefix closure π, without which all transmissions would have to have matching receipts, which would preclude the possibility of both ends being shut down simultaneously by a simultaneous transmission of two disconnect messages one at each end.

The rest of the expression denies the existence of any event v preceded by a distinct disconnect event u (one satisfying the predicate δ, assumed given) with u and v colocated. The effect of conjoining this denial with the two-way channel is to restrict the channel to that subset of it consisting of just those pomsets not having such a two-event sequence at either end. Using a global instead of a local diamond would have an almost identical effect, the one difference being that when the disconnect message is transmitted from end A, end B would appear to shut down just before receiving the disconnect message rather than just after, a rather fine point. If however we were to apply α in addition to π to the two-way channel, say for the sake of a more natural model, then end B could shut down well before receipt of the disconnect message because the receipt of the disconnect at B no longer has to be the earliest event at B that follows the transmission of that disconnect at A.

This expression improves significantly on our earlier[24,20,1] specifications of this object, which conveyed the same information (to within a minor bug or two) in some dozen lines of predicate calculus.

The temporal operator (implicitly an existential quantifier) could be dispensed with as follows. For languages one may define the predicate asserting the presence of a symbol or set of symbols s in a string as the set of all strings with this property, which we may write as $\Sigma^* s \Sigma^*$. Similarly the set of all finite pomsets containing s may be expressed as $\alpha(s \| \Sigma\dagger)$ (exercise). We may then take s in this expression to be the set $\bigcup\{((d \times E) \times e)((M \times E) \times e) | e \in E\}$ where $d \in M$ is the disconnect message, and Σ to be $(M \times E) \times E)$, which would give us the set of all pomsets containing an event strictly locally preceded by a disconnect message. The logic approach seems much simpler however in this case, and the temporal operator provides an additional measure of brevity and clarity.

This process provides an excellent example where algebra seems better suited than logic to one part of the problem but where logic apparently outdoes algebra in another. It is our thesis that *algebra and logic complement each other well in a process specification language.*

Note that it is not just brevity that is important here, but semantic clarity. While this is hard to judge objectively, we consider the above expression to be not merely short but also clear in what it expresses.

This two-way channel with disconnect generalizes very simply to a more general communication network, whose one-way channels form a subset $F \subseteq E^2$ of pairs of edges (excluding pairs (e, e)) of a graph describing the "topology" (connectivity) of an arbitrary communication net over a set E of channel ends.

$$\pi\underline{\lambda}(\mathop{\|}_{(a,b)\in F}((M^* \times a) \times ab)) \wedge \neg\underline{\diamondsuit}^{\pm}\delta$$

In this process a disconnect message from origin a to destination b shuts down a immediately after its transmission, and b immediately after its receipt. Thus n disconnect messages may shut down up to $2n$ ends. It is a worthwhile exercise to verify for this process as defined above that messages sent from a surviving end to an end shut down by some other end will be transmitted but not received, as expected.

§4 Systems of Communicating Processes

4.1 Informal Notions

There is a certain sense in which concurrency in a system of communicating processes (not necessarily sequential processes) can be thought of as conjunction. If we consider the set R of all possible worlds in which I might be performing action A, and the set S of all possible worlds in which you might be performing action B, then the system consisting of the set of all possible worlds in which we might jointly perform A and B is surely $R \cap S$.

This suggests that the process (set of behaviors) defined by a system of processes be taken to be the intersection of its constituent processes.

However there is something missing: nowhere have we made explicit any cooperation, communication, or coordination in the system. We must be assuming it is built into the components.

Yet this is not how components are described in component catalogs. No mention is made for example of a particular printed circuit board into which a given component is to be plugged. Instead components are described in isolation as self-contained processes.

We take care of this with the notion of a *utilization,* a function on processes. A system of processes is defined not as the intersection of its constituent processes but instead as the intersection of *the utilizations of* its constituent processes. A utilization is an application-dependent concept that contributes to a process the additional knowledge of how that process is related to its fellow processes.

This concept subsumes many familiar concepts, such as Kahn net semantics[18], and its extension by Brock and Ackerman to the nondeterministic case[17], along with simpler concepts such as the composition of binary relations, but extending also to analog domains, describing the behavior of a network of resistors in sufficient detail that one can infer say Thevenin's theorem. The sorts of components that can be used to build systems are exactly those we have treated above; thus we can take the CarCdr process defined in section 3.2, or the two-way-channel-with-disconnect, and wire it into a network of other processes to define a bigger system. We can then take say the orthocurrence or the star of the resulting system.

This is not done however by adding utilization as yet another operation on processes. Instead utilization is derived from the existing operations; in fact all we use are inverse homomorphisms and process intersection.

One last remark before diving into the formalism. The notion of system contemplated here is extremely general. It covers almost any situation involving sharing. The notion of sharing that should suggest itself readily to an electrical engineer is the sharing of component terminals by connecting them electrically. To a programmer sharing may mean shared variables in memory, or it may mean the sharing that occurs when the components of a flowchart are connected together: the exit of one command is shared with the entry to the next. To a physicist it may mean the variables of a physical system; thus when one body pushes another they share a force (to within sign), by Newton's law of equal and opposite reaction. To a mathematician it may be the sharing of variables between a system of equations, the sharing between two binary relations when they are composed, or the sharing that goes on between sections in a sheaf(25). *All* of these notions of sharing are contemplated in the following formal model of a system.

4.2 Translations and Restrictions

When a component is incorporated into a system, as when an integrated circuit is plugged into a printed circuit board, a correspondence is established between component events and system events. The kind of correspondence we shall consider takes the form of a function from the alphabet Σ' of component actions to the alphabet Σ of system actions. We shall call such a function a *translation* $t : \Sigma' \to \Sigma$ of alphabets. For example the event consisting of the appearance of 5 volts on pin 3 of an integrated circuit may translate to the appearance of Boolean 1 on trace 237 of the printed circuit board, this being the trace to which pin 3 is attached. In this case we would say that the value of the translation at $(5, 3)$ is given by $t(5, 3) = (1, 237)$. If pin 7 is also attached to this trace then we would want $t(5, 7) = (1, 237)$ also.

Given a translation $t : \Sigma' \to \Sigma$ from a component alphabet Σ' to a system alphabet Σ, and given a system behavior p (a pomset on Σ), it should be intuitively plausible that we can cut down or *restrict* p to just the part of it that involves the component. In fact merely deleting from p those events whose actions are not in the range of t would seem to do the job. We shall go two steps further than this. We shall rename the remaining actions to all be in the alphabet of the component. And when there is an ambiguity in this renaming caused by t not being injective (one-one) we shall "let them all win" by expanding each ambiguous event labelled x to a set of temporally incomparable events each labelled with one of the component actions mapped by t to x. Thus if $t(a) = t(b) = x$ then any event labelled x expands to two incomparable events labelled a and b respectively. Doing all this to p converts it into a pomset on Σ', that is, the restriction defined by $t : \Sigma' \to \Sigma$ is a function from pomsets on Σ to pomsets on Σ'. Since it acts locally on p, replacing each event by a pomset (actually a set), it must be a pomset homomorphism. This homomorphism is the restriction corresponding to t, which we denote ρ_t.

There is an elegant way to express ρ_t. Take the inverse of t, a function $t^- : \Sigma \to 2^{\Sigma'}$. Compose with t^- the inclusion of $2^{\Sigma'}$ into $\Sigma'\ddagger$ defined by the set-to-pomset coercion of section 2.2. That is, interpret the set $t^-(x) \subseteq \Sigma'$ as a pomset. We use t^- to also denote this composition, so $t^- : \Sigma \to \Sigma'\ddagger$. This makes t^- a function from atoms to pomsets; as such it determines a pomset homomorphism $t^{-+} : \Sigma\ddagger \to \Sigma'\ddagger$ mapping system behaviors to component behaviors. We take ρ_t to be t^{-+}.

It is easily verified that $\rho_{1_\Sigma} = 1_{\Sigma\ddagger}$ (where 1_X denotes the identity function on X) and $\rho_{t \circ s} = \rho_s \circ \rho_t$ (ρ preserves composition, albeit contravariantly). This property is central to verifying that process composition is associative. For those who like category language, it also establishes that ρ is a contravariant functor. In more detail, $\rho : \mathbf{Set} \to \mathbf{Set}$ is the functor on the category **Set** mapping the set Σ (regarded as an alphabet) to the set $\Sigma\ddagger$ of all finite pomsets on Σ, and mapping the function $t : \Sigma' \to \Sigma$ to the function $\rho_t : \Sigma\ddagger \to \Sigma'\ddagger$.

4.3 Systems and Utilizations

In the following we have in mind a system having the following features. Σ is the *system alphabet*, a global set of actions. For example the system may be a printed circuit board with set W of traces (printed wires), with $\Sigma = D \times W$ where D is a set of data values, e.g. $\{0, X, 1\}$. Thus a typical system action would be $(1, 237)$ denoting value 1 appearing on trace 237. I is an index set for the system components; its elements can be thought of as component names. Thus if the printed circuit board had 93 sockets for integrated circuits and other components then $|I| = 93$. In socket i we find a component whose process (set of possible behaviors) is referred to as P_i. The alphabet of P_i is Σ_i; thus if socket i has 14 pins then Σ_i might be $\{0, X, 1\} \times \{1, 2, \ldots, 14\}$. The correlation between the actions of component i and those of the system is established by a translation t_i of the kind contemplated in the previous subsection.

We reduce these notions to a formal definition as follows.

Definition 4.1. A *system* S is a family of pairs (P_i, t_i) over an index set I, where $P_i \subseteq \Sigma_i\ddagger$ is a process on Σ_i and $t_i : \Sigma_i \to \Sigma$ is a translation from P_i's alphabet to the *system alphabet.*

We then give the semantics of a system.

Definition 4.2. The *process realized by* S is $\{p \in \Sigma\ddagger \mid \forall i \in I[\rho_{t_i}(p) \in P_i]\}$.

Another way to express this uses the inverse homomorphism $\rho_{t_i}^- : 2^{\Sigma_i\ddagger} \to 2^{\Sigma\ddagger}$, which maps the process P_i to the process consisting of all pomsets in $\Sigma\ddagger$ that are mapped by ρ_{t_i} to a pomset in P_i. An equivalent expression for the process realized by the S in the above definition is then

$$\bigcap_{i \in I} \rho_{t_i}^-(P_i)$$

.

These inverse homomorphisms ρ_t^- are evidently the utilizations promised in the introduction to this section.

In category language, the functor mapping ρ_t to its inverse is the contravariant power set functor on **Set**. Composing it with the contravariant functor ρ yields a covariant functor on **Set** which maps translations to utilizations. A suitable name for it is the *utilization functor*, U_t.

We may obtain the notion of net as a special case of a system. Let D be any set (the data domain) and let C and C_i be channel sets for the system and for component i respectively. Then a *net* is a system for which $\Sigma = D \times C$, $\Sigma_i = D \times C_i$, and $t_i : \Sigma_i \to \Sigma$ has the form $t_i(x, c) = (x, f_i(c))$ where $f_i : C_i \to C$ is the i-th *connection* function. That is, $t_i = I_D \times f_i$, where I_D is the identity function on D.

As an example of a system (and of a net), consider a net with channel set $\{E, F\}$ and a component of the net with channel set $\{C, D\}$. Insert the component into the net via a connection mapping both C and D to E, thus connecting C and D together. Let the data domain be $\{0, 1\}$. Then the corresponding translation t satisfies $t(0, C)) = (0, E)$, $t((1, D)) = (1, E)$, etc., while the corresponding restriction ρ_t satisfies $\rho_t((1, E)) = (1, C) \| (1, D)$, $\rho_t((1, E)(0, E)(1, F)) = ((1, C) \| (1, D))((0, C) \| (0, D))$, etc.

At this point it is natural to ask, why interpret $\rho_t(x)$ as an unordered pomset when x is an atom? If $\rho_t(x)$ has more than one event, shouldn't we be making provision for the possibility that the component has a pomset containing these two events, but in one order or the other?

The answer is that any pomset of the component in which these two events are comparable is inconsistent with the system requirement that the two events occur simultaneously. Even when a process has the two events occurring in both possible orders in different pomsets, this is not considered evidence that they can occur simultaneously in that process. The process must have a pomset in which they occur incomparably, and only that pomset will be recognized as being able to cause the system action x resulting from the identification by t of the two component actions. Hence $\rho_t(x)$ as an unordered pomset is the only pomset permissible as the "cause" of x.

A brief remark on generality. We noted in the introduction to this section that our notion of system was very general. To realize the generality we promised however requires a more general notion of object realized by a system than a set of pomsets. Hence in the definition of this object as the set of all pomsets on Σ whose restrictions to the components are in those components, other sources of elements of that set besides pomsets must be possible. However we assumed in defining ρ_t that we were dealing with pomsets, when we resolved ambiguities using unordered pomsets. In the absence of pomsets some other resolution of the ambiguity must be found; one approach that will cover most cases of interest is to require the translations to be one-one. This is what happens with sheaves[(25)], discussed below.

It is intuitively plausible that the structure of systems assembled hierarchically in this manner is associative in the sense that the structure could be flattened out to a single level. It is an easy exercise to show that this is indeed the case, knowing only that the map from translation to utilization is a functor and that utilization distributes over conjunction.

4.4 Encapsulation.

Thus far we have only shown how to build big processes from smaller ones. We would also like to be able to shrink these big processes back down by hiding or packaging or *encapsulating* that part of the computation that is to be considered internal and leaving only the external part visible.

This need is met using restriction. The idea is to consider the external actions of the system as just the actions of another component. In our printed-circuit-board example, although PC boards usually make outside contact via edge connectors and headers on the edges of the board, occasionally the connection is made by an ordinary socket located alongside the component sockets. Topologically at least there is nothing to distinguish the outside world from just another (large!) component. We already have a mechanism for relating the actions of such a component to the system actions, namely translation. Hence for encapsulation purposes it suffices to use an ordinary translation, which we may call the external translation t_e. Then the external behavior of the process P_S realized by system S is $\rho_{t_e}(P_S)$, the result of restricting P_S to the "external" component e.

We now have enough machinery to build up systems hierarchically. At each stage form a system behavior as the intersection of the utilizations of the system components, then restrict that behavior to the external component. The result can then be used as a component of another system. Or it can be used as an argument to any of the pomset operations of section 2.

While any processes may be combined in this way, the results may be counterintuitive unless those processes are prefix closed and augment closed. The role of prefix closure is to ensure that a system can get part way through a computation even when some components refuse to carry the computation through to the end. This is essential for a natural model of how components communicate. The role of augment closure is to permit a process that orders two events to share both events with a process that does not order them, the idea being that the second process should have no objection to observing the order demanded by the first. Augment closure should be applied in this context whenever incomparability means that order does not matter. If however the meaning of two events being incomparable is that they must happen simultaneously, in the sense that neither may precede the other, then augment closure should not be applied, so that these events remain incomparable. For example the process may be an exam, where it is unacceptable to have one student leave the exam before another arrives. In this situation the two events must remain incomparable. Hence while prefix closure seems to be mandatory for system components, augment closure need not be when incomparability is being interpreted as simultaneity or overlap.

In the broader context of all pomset operations, it is not desirable to make prefix closure mandatory. For example, although $\pi(pq)$ is equal to $\pi(p\pi(q))$ it is not in general equal to $\pi(\pi(p)\pi(q))$. The difference between $p\pi(q)$ and $\pi(p)\pi(q)$ is that in the latter, events of q do not wait for all the events of p (e.g. p may abort and then q may start), unlike the case in the former. This consideration dictates making explicit all uses of prefix closure. It is not good enough just to declare in advance that all processes and their associated operations are prefix closed. On the other hand we do have $\pi(p)\pi(q) = \pi(\pi(p)\pi(q))$. That is, concatenation preserves prefix closure. Hence prefix closure is at least consistent with concatenation, if not always desirable.

4.5 Applications of Systems.

This approach to the semantics of systems is very far-reaching. It can model composition of binary relations, Kahn semantics[18], composition of processes via arbitrary nets, connection via a bus or ethernet where messages may be broadcast by any process attached to the bus or ethernet to any other such process (since there is no limit to how many processes may share a channel), and analog circuits such as a net of resistors (since each of the events and the actions can be a continuum, modulo replacing $\Sigma\ddagger$ by pomsets on Σ having up to $\aleph_1$ vertices when defining the semantics of systems). (For the case of resistors, and many similar analog systems, where impedances are relevant, it appears necessary to represent the junctions between resistors as components in order to account for Kirchhoff's law.) It can also be extended to deal with real-time systems, where events are related not merely by their temporal order but by the amount of delay between them; the technique[20] is to generalize the notion of partial order to the notion of transitive matrix over a semiring. The utilization approach handles all these diverse systems by being defined quite abstractly, without regard for the details that characterize any one of these kinds of systems. Our paper[20] contains examples and more details.

Sheaves. Monteiro and Pereira[25] present a model of concurrency based on the algebraic geometry notion of a sheaf. To define a sheaf first define a *presheaf* to be a contravariant functor F from a category with objects the open sets of a topological space X and morphisms their inclusions, to the category **Set**. Let ρ^U_V for $V \subseteq U$ denote the image under F of the inclusion from V to U, called the *restriction map* from $F(U)$ to $F(V)$. For any $s \in F(U)$, called a *section over* U, abbreviate $\rho^U_V(s)$ to $s|_V$, the *restriction of* s to a section over V. Then a *sheaf* on X is a presheaf on X such that if U is an open set of X having as open cover a family $\{V_i | i \in I\}$, and $\{s_i \in F(V_i) | i \in I\}$ is a family such that for all $i, j \in I$ $s_i|_{V_i \cap V_j} = s_j|_{V_i \cap V_j}$, then there exists a unique $s \in F(U)$ such that $s|_{V_i} = s_i$ for all $i \in I$.

In the context of systems of processes the idea is that a section s_i is a behavior of the i-th component or *location* V_i of the system U, where the locations are organized hierarchically according to the inclusion structure of the V_i's. $s|_V$ restricts the behavior s of location U to the sublocation $V \subseteq U$. When behaviors can be found for all locations such that the behaviors agree pairwise on the common portions of those locations, then there exists a unique system behavior that agrees with all the location behaviors. System dynamics is provided for by Monteiro and Pereira by assigning to the objects in the image of F the structure of monoids.

This approach has some important points in common with our approach, as well as some important differences. The points of correspondence are as follows. In place of a category of open sets of a topological space X and their inclusions we use the category **Set** whose objects (sets) provide our alphabets and whose morphisms (functions) our translations. By using arbitrary functions rather than inclusions we provide for action renaming (via non-inclusions) and "short-circuiting" (via non-injections). The concept of restriction has the same significance for both approaches, and is a contravariant functor in both cases. The explicit concept of section agreement on intersections, which is the essence of a sheaf, appears only implicitly in our approach, by virtue of the possibility of overlap between the $t_i(\Sigma_i)$'s. We do not require the $t_i(\Sigma_i)$'s to cover Σ, but then neither do we require that the system behavior consistent with a family of component behaviors be unique. In place of monoids we use processes made up of pomsets, which come with a built-in solution to the problem of non-injective translations and which also have the several advantages cited for them in the introduction, not achievable with monoids.

Our approach may be massaged into closer correspondence with the sheaf approach by replacing the Σ_i's with their images $t_i(\Sigma_i)$ as subsets of Σ, and the t_i's with the corresponding inclusions from $t_i(\Sigma_i)$ into Σ, provided the t_i's are injective and the $t_i(\Sigma_i)$'s cover Σ (easily arranged by adding a dummy process to complete the cover). There is no loss of generality in requiring these subsets to be open since in the absence of other requirements Σ can be equipped with the discrete topology (all subsets open). The information in the P_i's is now not accessible since the t_i's are gone, so, taking our sheaves to be sheaves of processes rather than of monoids, we move the P_i's into the restriction functor by having ρ map $t_i(\Sigma_i)$ to $t_i^+(P_i)$ where t_i^+ is the extension of t_i to a pomset homomorphism. The remaining step is to have ρ map the other subsets of Σ, or at least the arbitrary unions and finite intersections of the $t_i(\Sigma_i)$'s, to appropriate subsets of $\Sigma\ddagger$, in order to make ρ into a sheaf, though we have not yet worked out the appropriate assignment.

Unfortunately this massaging destroys what we feel is one of the nice features of our approach, the concept of translation between alphabets, which aptly describes how components are integrated into systems. We feel that our approach is better presented in terms of translations than sheaf-theoretically. Also topology and continuity play no role in our model at present, allowing us to dispense with topological spaces. On the other hand a possible use for the sheaf approach in our theory might be to cater for least fixed points of continuous functions, using a nondiscrete topology on Σ, though we do not currently understand this well.

Petri Nets. Petri nets(7) are easily modelled in our framework. Each place fed by m transitions and feeding n transitions, whether or not all distinct, may be modelled as a process with m inputs and n outputs. Take the alphabet for all places to be $N \times \{I, O\}$ consisting of pairs (i, d), a port-direction pair, the port being a number and the direction being I (in) or O (out). Let $\mathbf{m}$ denote the process $\{0, 1, \ldots, m-1\}$ consisting of m atomic pomsets, and similarly for $\mathbf{n}$, and let T (for token) denote $(\mathbf{m} \times I)(\mathbf{n} \times O)$. Then T is a process whose typical pomset is the string $(i, I)(j, O)$ of length 2, being the concatenation of the arrival of a token on input i with the departure of that token on output j. For place/transition nets, where tokens may flow concurrently through a place, form $T' = \alpha(T\dagger)$. For condition/event nets, where places may hold only one token at a time, take $T' = T^*$. Then the process modelling this place is $\sigma(\pi(T'))$, where $\sigma(P) = \pi(P^-)^-$ is the suffix closure of P. (The role of suffix closure is to permit computations to begin with tokens having already arrived at some places.)

A Petri net is then a system whose components are places, whose system alphabet is just the set of transitions of the net, and whose translations t_k (for place k) map (i, I) or (j, O) to the transition to which input i or output j of place k is connected. The system events then correspond to firings of transitions. The process realized by the system consists of the usual behaviors of a Petri net[6,7], each consisting of a transition-labelled poset (i.e. pomset) of transition firings. Note that the case $|\rho_{t_k}(x)| > 1$, i.e. transitions x connected to more than one port of the same place k, is handled correctly.

Kahn Nets. Kahn nets[18] are defined in terms of least fixed points of a system of equations involving variables and continuous functions on a domain $D^* \cup D^\omega$. Kahn nets are only defined for deterministic processes; however the elegance of Kahn's model demands an answer to the question of what is the correspondence between his model and ours.

The answer is that there is a translation from Kahn's model to ours in which Kahn's processes become our processes, at least for finite nets. The key observation in this translation is that a solution to Kahn's equations is translatable to a pomset if and only if the solution is minimal. This is because the translation finds a cycle in the order if and only the solution is not the least one. This property has been independently observed by Nagatsugu Yamanouchi [private correspondence]. We will say more about this correspondence elsewhere.

§5 Conclusions

The notions of event in a partially ordered behavior, and process as a set of behaviors, have proved both natural and workable in the specification of concurrent systems. Techniques from such diverse approaches as formal language modelling of concurrency, the Brock-Ackerman generalization[17] of Kahn's fixpoint approach to nets [18] and Pnueli's temporal logic[22,23] come together naturally in this framework. A variety of notions are handled gracefully: continuous time (we may take the reals for a vertex set), continuous data (we may take the reals for a data domain), channels with asynchronous ends (handled by orthocurrence), hierarchical composition (a corollary of our algebraic approach, as well as of the possibility of hierarchy in systems), multi-access channels such as ethernets and buses (since channels in nets need not be directional), and events with real-time constraints.

We have not developed any formal rules for reasoning with this approach. However since our approach has substantial overlap with other approaches that have seen much logical development, we are optimistic about the prospects of filling this gap, partly with recycled algorithms and partly with new ones.

We have said nothing here about probabilistic processes. It seems to us however that the existing techniques that have emerged for sequential processes ought to carry over very smoothly to this framework. One can imagine imposing some notion of measure on events for the purpose of assigning probabilities to them, and otherwise maintaining much of the compositional part of the theory unchanged. The idea of flow of probability, under an appropriate measure, through a net makes just as much sense as flow of data.

Finally, we have generally omitted proofs. While some of our claims really have the status of postulates, e.g. the definition of utilization, there are others that demand proof, e.g. existence and uniqueness of the extension h^+ of any function $h : \Sigma \to A$ for any pomset algebra A, the equivalence of utilization-based composition to Kahn composition for the case of determinate processes, and adherence of utilization composition to the expected behavior of a variety of difficult cases, including all known Brock-Ackerman-style anomalies.

ACKNOWLEDGMENTS. Haim Gaifman, Ross Casley, and the members of my concurrency modelling seminar provided valuable criticism and ideas. The motivational section in the introduction was included at the request of the referees, one of whom also contributed several helpful arguments for and against pomsets that we have gratefully taken into account.

REFERENCES

[1] Pratt, V.R., Some Constructions for Order-Theoretic Models of Concurrency, Proc. Conf. on Logics of Programs, Springer-Verlag LNCS 193, Brooklyn, 1985.

[2] Denvir, T., W. Harwood, M. Jackson, and M. Ray, **The Analysis of Concurrent Systems**, Proceedings of a Tutorial and Workshop, held Sept. 1983, Cambridge University, LNCS 207, Springer-Verlag, 1985.

[3] Barney, C., Logic Designers Toss Out the Clock, Electronics **58**, 49, 42-45, Dec. 9, 1985.

[4] Gischer, J., **Partial Orders and the Axiomatic Theory of Shuffle**, Ph.D. Thesis, Computer Science Dept., Stanford University, Dec. 1984.

[5] Greif, I., Semantics of Communicating Parallel Processes, Ph.D. Thesis, Project MAC report TR-154, MIT, September, 1975.

[6] Best, E., C. Fernandez, and H. Plünnecke, Concurrent Systems and Processes, Final Report on the Foundational Part of the Project BEGRUND, GMD-Studien Nr.104, GMD, Sankt Augustin, FDR, March 1985.

[7] Reisig, W., **Petri Nets: An Introduction**, Springer-Verlag, 1985.

[8] Winskel, G., Events in Computation, Ph.D. Thesis, CST-10-80, Dept. of Computer Science, University of Edinburgh, 1980.

[9] Winskel, G., A New Definition of Morphism on Petri Nets, Proc. CMU/SERC Workshop on Analysis of Concurrency, Springer-Verlag LNCS 197, Pittsburgh, 1984.

[10] Winskel, G., Categories of Models for Concurrency, Technical Report no. 58, University of Cambridge, England, undated (rec'd Dec. 1984).

[11] Pinter, S.S., and P. Wolper, A Temporal Logic to Reason about Partially Ordered Computations, Proc. 3rd ACM Symp. on Principles of Distributed Computing, 28-37, Vancouver, August 1984.

[12] Lamport, L., On Interprocess Communication, DEC Systems Research Center, Report No. 8, 1985.

[13] Van Benthem, J.F.A.K., **The Logic of Time**, D. Reidel, 1983.

[14] Whitrow, G.J., **The Natural Philosophy of Time,** 2nd ed., Oxford University Press, 1980.

[15] Mazurkiewicz, Traces, Histories, Graphs: Instances of a Process Monoid, Proc. Conf. on Mathematical Foundations of Computer Science, Springer-Verlag LNCS 176, 1984.

[16] Pratt, V.R., On the Composition of Processes, Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages, Jan. 1982.

[17] Brock, J.D. and W.B. Ackerman, Scenarios: A Model of Non-Determinate Computation. In: **Formalization of Programming Concepts**, J. Diaz and I. Ramos, Eds., Springer-Verlag LNCS 107, New York, 1981, 252-259.

[18] Kahn, G., The Semantics of a Simple Language for Parallel Programming, IFIP 74, North-Holland, Amsterdam, 1974.

[19] Kahn, G. and D.B. MacQueen, Coroutines and Networks of Parallel Processes, IFIP 77, 993-998, North-Holland, Amsterdam, 1977.

[20] Pratt, V.R., The Pomset Model of Parallel Processes: Unifying the Temporal and the Spatial, Proc. CMU/SERC Workshop on Analysis of Concurrency, Springer-Verlag LNCS 197, Pittsburgh, 1984.

[21] Milner, R., Calculi for Synchrony and Asynchrony, Theor. Comp. Sci. 25 (1983), 267-310.

[22] Pnueli, A., The Temporal Logic of Programs, 18th IEEE Symposium on Foundations of Computer Science, 46-57. Oct. 1977.

[23] Gabbay, D., A. Pnueli, S. Shelah, and J. Stavi, On the Temporal Analysis of Fairness, Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages, Jan. 1980, 163-173.

[24] Pratt, V.R., Two-Way Channel with Disconnect, in Denvir et al[(3)], section 3.1.3. (1983)

[25] Monteiro, L.F., and F.C.N. Pereira, Outline of a Sheaf-theoretic Approach to Concurrency, Proc. IEEE Symp. on Logic in Computer Science, Boston, July, 1986.