On system executions and states

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

March 12, 1995

Abstract

An approach for modeling dynamic situations where several processes execute concurrently is described here. We address basic issues such as what is an 'event'? What is a 'state' of a system? How should time be represented? etc. In this, we use the concepts 'system execution' and 'pomset' defined by Lamport and Gischer.

The notion of a system execution is extended in such a way that a state is just a peculiar system execution, one in which no event precedes another—it is the description of a moment. This allows to demonstrate within a single framework the duality between two approaches to modeling concurrency: the one based on events and the other on states.

1 Introduction

The notions of a *structure* and a *model* were proved to be very useful in mathematics, but it is not obvious how to adapt them to represent an ever changing universe. A structure $S = (A, F)$ consists of a 'universe of discourse' A, which is just a set, and a collection F of functions and relations defined on A. A language, L, is interpreted in S, by associating with each function and relation symbol of L the corresponding function and relation in F. All of this is well known, and we owe it to A. Tarski the recursive definition of 'truth' which tells when a sentence of L is satisfied in S. We say that S is a *model* of the set of sentences it satisfies.

Sometimes models can be used, in a quite naive way, to describe real-life situations. 'Poushka is a beautiful dog' holds in S if there is an object $p \in A$ which interprets the constant '*Poushka*' and there are predicates *Dog* and *Beautiful*, and the sentence

$$Dog(Poushka) \wedge Beautiful(Poushka)$$

is satisfied in S. However one soon discovers that to describe activity and change this approach is not sufficient. For example, to model 'Poushka barks at anyone approaching the fence' it seems appropriate to introduce *event occurrences* as well as constant objects. Unlike the constant objects, the event occurrences are situated in time, and the temporal precedence relation can be applied to them. Our example may be approximately rendered as: Any event x representing someone approaching the fence is followed by an event representing Poushka barking.

A careful distinction should be drawn here between *occurrences* (of events) and *events*. A particular bark at a certain time and location is an occurrence of the general bark event. Unlike occurrences, events are not situated in space and time and it is not meaningful to say that one event follows another.

This distinction was made by K. R. Popper (Section 22 of [1959]) who defined 'event' to denote "what may be *typical or universal* about an occurrence". An event is a "class of occurrences which differ *only* in respects of the individuals (the spatio-temporal position or regions) involved".

If we seriously want to take an event to be a class of occurrences which are similar in all respects except for their spatio-temporal position, we run into the difficulty that it is very rare to see two occurrences that are really separated only by their spatio-temporal positions. For example, if we render the event 'Poushka barks' as the set of barking occurrences, we must admit that some of the barks are louder than the others, and a very careful examination might reveal that no two bark occurrences have exactly the same characteristics. Despite this and other difficulties we will try to give a formal definition for 'event' in the context relevant to computer science. (Some writers use the terms *event* and *event type* instead of *occurrence* and *event*; this seems very natural, and sometimes quite inconsistently we may write 'events' where we should have written 'occurrences'.)

The primality of the notion of an 'occurrence' was also stressed by B. Russel and N. Wiener [1914] who showed how the spatio-temporal position is in fact definable from the event occurrences. Thus, instead of saying 'Poushka barks at noon', they would look at the collection T of event occurrences which defines the moment 12:00 (this collection may include someone reading her watch and seeing that it is 12:00 etc.), and say that T includes a bark occurrence that is associated with Poushka. A formal development of this idea (following Wiener [1914]) is given in Section 2.

The main aim of this paper is to advocate a framework for describing distributed, concurrent processes. This framework is quite general and may be of interest to readers looking for precise domains for the representation of temporal knowledge. Knowledge-based systems have become increasingly important in the past decade. Expert systems have become commonplace in a myriad domains through the use of sophisticated shells and other tools. The terms 'state,' 'process,' 'event,' and 'interval' are defined and redefined while work from logicians and philosophers on 'modal,' and 'temporal' logics have worked their way into the common vocabulary of knowledge representation. Much of the language and many of the techniques being applied here are the same as those which have, for perhaps a longer time, been actively exploited in another area: concurrent systems. The precise description of such systems is the first requirement for establishing methods for verification and testing of concurrent programs. It is precisely in the non-deterministic temporal ordering of the executions of these programs where the difficulty arises in their semantics.

This paper addresses some of the central concerns of providing precise temporal models for discussing concurrent and distributed systems. Just as a planning system must represent and reason about order among the steps and objectives, so a program verifier must represent and reason about the permissible orders of the execution of instructions. Especially important is the establishment of semantics for relating the written program with the individual executions of the program.

For these aims, we integrate two well known notions: system executions and pomsets (see (Lamport [1986], and Gischer [1989] and see also V. Pratt [1986]). Many other investigators were aware of the importance of dealing with occurrences and temporal relations, and if we mention here only these two concepts it is only because of our desire to concentrate on what seems to be a useful combination of the system execution and pomsets models. Let us give first an informal description of these notions and the way we combine them. The formal definitions are given in Sections 2 and 3.

A model is not only an abstract description of reality, but also a simplification that disregards some non-essential features. Thus, for many applications in computer science, the exact beginning and ending time of each event is not needed. What really counts is only the *precedence* relation among the events. This relation is a partial order (two occurrences that 'overlap' in time are incomparable). In its purest form, this approach is pursued by L. Lamport [1986] who defined a (global-time) system execution to be a structure $S = (E,<_{\mathcal{S}})$ where $E_{\mathcal{S}}$ is a set of occurrences and $<_{\mathcal{S}}$ is the precedence relation on $E_{\mathcal{S}}$. Two requirements are imposed on $<_{\mathcal{S}}$.
(1) $<_{\mathcal{S}}$ can be interpreted as the partial order on a set of intervals of some linear order. That is, we can think of the occurrences in $E_{\mathcal{S}}$ as intervals and $I <_{\mathcal{S}} J$ means that I ends before J begins. (A formal, more exact definition is given in Definition 2.2).
(2) For every terminating event $I \in E_{\mathcal{S}}$, there are only finitely many events in E that begin before I ends.

Gischer [1989], in defining pomsets, is not so much concerned with the properties of the precedence relation, as with the character of each event occurrence in $E_{\mathcal{S}}$. Thus a pomset (partially ordered multiset) is a structure $(E_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}})$ where $<_{\mathcal{S}}$ is a partial order on $E_{\mathcal{S}}$, and $F_{\mathcal{S}}$ is a collection of unary predicates on $E_{\mathcal{S}}$ (a unary predicate is a one place predicate).

We suggest here the following combination and extension of these two notions (in Section 3). A *system execution* is a two-sorted structure $\mathcal{S} = (E_{\mathcal{S}}, C_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}})$ where $E_{\mathcal{S}}$ is the set of occurrences, $C_{\mathcal{S}}$ is the set of constant objects, $(E_{\mathcal{S}}, <_{\mathcal{S}})$ is a global-time system execution in the sense of Lamport, and $F_{\mathcal{S}}$ is a collection of functions and predicates on $E_{\mathcal{S}} \cup C_{\mathcal{S}}$.

We find it useful to have both event occurrences and constant objects in the same structure. For example, to describe an execution of a program, one can put in $C_{\mathcal{S}}$ the program text which is a constant object, and have in $F_{\mathcal{S}}$ a function that associates to each operation execution $e \in E_{\mathcal{S}}$ the command in the program that e is executing. (This seems to us more natural than having separate predicates for each command in the program.) Not only functions from $E_{\mathcal{S}}$ into $C_{\mathcal{S}}$ are useful, but functions from $E_{\mathcal{S}}$ to $E_{\mathcal{S}}$ as well. For example, in describing the operation of a queue, one may use a function that associates to each dequeue event its corresponding enqueue event. In our experience, unary functions and predicates suffice, and the functions from $E_{\mathcal{S}}$ to $E_{\mathcal{S}}$ that arise in practice are *regular* or *bounded* (see Definition 2.6).

A concept of central importance in this work is that of a 'state'. Traditionally, a state is a function which associates with each variable a value. In this approach, the total reality is seen as a *sequence* of states. If however we adopt system executions as the basic means of describing reality, and if a state is nothing else but a reality with a very short time span, then system executions (of a special kind) should also be used to represent states. This will be developed formally in Section 3, Definition 3.1. The identification of states with special kinds of system execution will enable us to decompose any system execution into a sequence of states.

In ending this introduction, let us make some remarks about the global-time assumption. The global-time assumption is made in defining a system-executions to be a structure for which the partial order $<_{\mathcal{S}}$ on $E_{\mathcal{S}}$ can be represented as the partial order on a set of intervals of some *linear* order. This linear order is intuitively a representation of the time axis, and the existence of a linear time axis is not an obvious assumption. Physicists argue that linear time is not a reasonable assumption (see, for example, A. A. Robb [1921], cited by M. Capek [1976]), but in the area of computer science, as in everyday life, global-time is usually assumed. This should not be confused with the global-clock assumption, which is quite a strong assumption, especially when dealing with concurrent processes located at different cites and working independently of each other. (This assumption stipulates that the processes have instantaneous access to some common clock, and can use it to synchronize their actions). Nor should global-time be confused with serializability which claims that the event occurrences are, or can be thought to be, linearly arranged in time. Global-time is the metaphysical assumption of a time axis over which the events occur.

L. Lamport [1986] proposes a different treatment of temporal relations in which global time is not assumed: his approach involves two relations $\rightarrow$ and $\dashrightarrow$ on the event occurrences, and a short list of axioms (see Section 2.2). The reader who wishes to do so can adapt all the definitions and discussions of this paper to the framework of Lamport. This results in a wider collection of system executions and the global-time assumption can be expressed as $\forall x, y(x \not\dashrightarrow y) \Rightarrow (y \rightarrow x))$.

We do not take this general approach here, for simplicity and because of our belief in the soundness of the global time assumption in the realm of computer science. We believe that, at least as far as the correctness of protocols and algorithms is concerned, the assumption of global time can safely be made. That is, if an algorithm is proved correct with the global time assumption, then it will also work correctly in the more general framework proposed by L. Lamport. This belief has motivated the work by Ben-David; Abraham, Ben-David and Magidor; and Abraham, Ben-David and Moran, cited in the References.

Surprisingly, though we are assuming global-time here, our work naturally leads to structures where it does not hold. Its negation and the relations $\rightarrow$ and $\dashrightarrow$ appear naturally, not as a result of some physical intuition, but as means of describing the common features of a class of system executions (in which the global time axiom hold). Following the work of Ben David [1988] and Gaifman[], we define in Section 5 the notion of a 'core of a system-execution'. Global-time is not necessarily valid in these structures. Thus the general axioms of Lamport turn out to be meaningful even for those rejecting his distinction between $x \not\longrightarrow y$ and $y \dashrightarrow x$.

It is with the help of cores of system executions that we try to define the general notion of an 'event' (event type) and that of 'concurrent events'.

2 On Pomsets and system executions

In this section we give the original definitions of these notions which are so basic to our discussion, and explain why we found it desirable to combine and modify them. System executions were defined by Lamport [1986], and Pomsets were defined by Gischer [1984] and [1989]. As the notion of an 'interval-order' is a main ingredient in Lamport's definition, we discuss it first.

2.1 Interval-orders

Since any experience has some duration, the notion of a *time-interval* is prominent in any analysis of 'time'. It is natural to identify the precedence relation on event occurrences with the corresponding relation on the intervals of the real line (seen as the time axis). Thus a characterization of partial orders of intervals is useful for our investigation.

Definition 2.1 Given a linear order, $(L, <)$, define two relations $\longrightarrow_L$ and $\dashrightarrow_L$ on the intervals of L (nonempty convex subsets). For intervals A, B

1. $A \longrightarrow_L B$ iff $\forall x \in A, \forall y \in B \ (x < y)$.

2. $A \dashrightarrow_L B$ iff $\exists x \in A, \exists y \in B$ ($x < y$ or $x = y$)

Observe that $\longrightarrow$ is irreflexive and transitive, and $A \dashrightarrow B$ iff $\neg B \longrightarrow A$. Hence any one of these relations suffices to define the other. (L. Lamport [1986] is not assuming a linear ordering of the time axis, and in his work $\longrightarrow$ and $\dashrightarrow$ are not in general definable from each other. It may be remarked that even though $\longrightarrow$ and $\dashrightarrow$ can be defined from each other, we find it very convenient to actually have them both at hand.)

The resulting partial order $\longrightarrow$ on intervals is extensively studied in Fishburn [1987], and the earliest reference there to this subject is Wiener [1914]. In this paper N. Wiener follows the suggestion of B. Russell and defines (what we call today) interval-orders.

Definition 2.2 An irreflexive and transitive relation R on E is called an *interval-order* iff

$$\forall a, b, c, d \in E(aRb \wedge \neg cRb \wedge cRd \Longrightarrow aRd) \tag{1}$$

We shall call statement (1) the *Russell-Wiener* axiom. It is easy to see that for any linear order $(L, <)$, it holds for the relation $\longrightarrow_L$ on the intervals of L.

To understand this formula, define simultaneity as the relation of two events that neither one precedes the other. So if a precedes b and b is simultaneous with c, and c precedes d then a precedes d.

A *representation* of an interval-order (E, R) is a linear order L and a map μ of E into intervals of L such that for all $a, b \in E$, aRb iff $\mu(a) \longrightarrow_L \mu(b)$.

Theorem 2.3 (Russell and Wiener) *Any interval-order has a representation.*

The importance of this theorem is in showing that this concept, interval-order, really captures the intuitive idea we have about the precedence relation on events.

Proof. [Wiener 1914] Wiener's proof follows Russell's intuition which relates a moment to the collection of all epochs (time intervals) in which it occurs. So let be given an interval-order (E, R). Define $X \subseteq E$ to be a *moment* iff X is a maximal (under inclusion) antichain (X is an antichain iff $\forall x \neq y \in X(\neg xRy$ and $\neg yRx)$). Then let L be the collection of all moments, and define for $X, Y \in L$, $X < Y$ iff $\exists e \in X, \exists f \in Y$ (xRy).

It follows that $<$ is a linear order on L (transitivity is an easy consequence of the Russell-Wiener formula, and linearity follows from the maximality of moments). Now define $\mu(e)$ for $e \in E$ to be the collection of all moments X such that $e \in X$.

We claim that

1. $\mu(e)$ is convex (an interval in L), and
2. for all $e, f \in E$, eRf iff $\mu(e) \longrightarrow_L \mu(f)$.

To prove the convexity, assume $X < Y < Z$, and $X, Z \in \mu(e)$. Then $e \in X$ and $e \in Z$. We will prove that $Y \in \mu(e)$. Assume for the sake of a contradiction that $e \notin Y$. By maximality of Y, there is $y \in Y$ such that eRy or yRe. Assume for example that eRy. Since $X < Y$ there are $y' \in Y$ and $z \in Z$ be such that $y'Rz$. Then the Russell-Wiener formula implies that eRz, and this contradicts the fact that the moment Z is an antichain.

In order to prove the second part of the claim, assume first that eRf. Let be given any $X \in \mu(e)$ and $Y \in \mu(f)$. Then $e \in X$ and $f \in Y$ by definition of μ, and so $X < Y$ by definition of $<$. Thus $\mu(e) \longrightarrow_L \mu(f)$. Next, let us see that if $\neg eRf$ then $\mu(e) \not\longrightarrow_L \mu(f)$. Assume $\neg eRf$. If fRe then $\mu(f) \longrightarrow_L \mu(e)$, as proved above, and hence $\mu(e) \not\longrightarrow_L \mu(f)$ by the transitivity and ireflexivity of $\longrightarrow_L$. If e and f are incomparable under R then a maximal antichain exists which includes both e and f. And again this shows $\mu(e) \not\longrightarrow_L \mu(f)$.

This proof is the one given by Wiener [1914]. The proof in Fishburn [1987] is different and more constructive.

2.2 Lamport's structures

Lamport offers a formalism for the analysis of distributed systems that is based on precedence relations among operation executions. His language consists of two binary relations $A \longrightarrow B$ for 'strong precedence', and $A \dashrightarrow B$ which can be read as 'A can causally affect B'.

Lamport's Axioms for strong and week precedence relations.

1. $\longrightarrow$ is irreflexive and transitive.
2. $a \longrightarrow b$ implies $a \dashrightarrow b$, and $\neg b \dashrightarrow a$.

3. $a \longrightarrow b \dashrightarrow c$ implies $a \dashrightarrow c$. And similarly $a \dashrightarrow b \longrightarrow c$ implies $a \dashrightarrow c$.

4. $a \longrightarrow b \dashrightarrow c \longrightarrow d$ implies $a \longrightarrow d$.

The following structures were defined by Lamport [1986] who called them 'system executions'.

Definition 2.4 We say that $(E, \longrightarrow, \dashrightarrow)$ is a *Lamport's* structure iff it satisfies the axioms listed above and moreover it satisfies the finiteness property: for every $x \in E$ the following set is finite.

$$\{y \in E \mid \neg x \longrightarrow y\}$$

Remark that the finiteness axiom is stronger than the one which states that for any $x \in E$ there are only finitely many $y \in E$ with $y \longrightarrow x$. Lamport [1986b] distinguishes between terminating and nonterminating events and the finiteness axiom is only stated for terminating events; but we prefer to limit here our discussion to terminating events.

An axiom of much interest is the *global-time* axiom.

The Global-Time Axiom

$$\text{For every } a \text{ and } b \ a \longrightarrow b \text{ iff } \neg b \dashrightarrow a.$$

The global-time axiom is not a consequence of Lamport's four axioms. In this paper the global-time axiom is accepted.

There are two other axioms that are of interest. They were investigated by Anger [1988]

Anger's axioms

1. For all x $x \dashrightarrow x$, and

2. $u \dashrightarrow v \longrightarrow x \dashrightarrow y$ implies $u \dashrightarrow y$.

2.3 Gischer's Pomsets

Definition 2.5 A *pomset* is a structure $\mathcal{P} = (E_{\mathcal{P}}, <_{\mathcal{P}}, F_{\mathcal{P}})$ where

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

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

We tend to view a pomset as a description of reality, one world out of many possible, rather than a prescription of how things must be. Thus $<_{\mathcal{S}}$ is just the precedence relation. Other writers (Gischer and Pratt for example) seem to view $<_{\mathcal{P}}$ as the *causality* relation. In our opinion, causality is a very complicated relation which depends on the theories involved in explaining the phenomena, and thus we opt for the simpler and neutral precedence relation.

2.4 Why another definition?

These two concepts, system executions and pomsets, complement each other: The system executions precise the type of the precedence partial order which is left unspecified in the pomset notion, and the pomsets allow for further (monadic) information about the event occurrences beside their relative temporal position. So it makes sense to combine these concepts, and we shall do so with the addition of more features which are described in short now (we continue to call these structures system executions).

First, we feel the need to add to the set of event occurrences, a set of 'fixed objects' which represent that part of the system execution that does not change in time. For example, if we are modeling executions of an algorithm, then the program itself which is executed by the processes is a fixed object, and there is a map that assigns to each operation execution its control place in the program. Thus our structures are basically two sorted structures that contain events and fixed objects.

Second, experience with protocols and correctness proofs has taught us the advantage of supplementing system executions with unary functions defined on the set of events and the set of fixed objects. So we suggest to enrich the definition of the structures that originally contained unary predicates with functions and arbitrary predicates. This has the obvious advantage of allowing a more flexible mode of expression, but has the disadvantage of entailing highly undecidable theories. The 'right' definition will be found after suitable experience is acquired, but we feel that the unary predicates are not enough.

For example, in describing the behavior of Poushka, the function that assigns to each event of barking its cause may be of help. Denoting this function Ω, $\Omega(b)$ is the event that caused b. Naturally, $\Omega(b) \not>_{\mathcal{S}} b$ for each barking event occurrence b. Here $x \not>_{\mathcal{S}} y$ is the negation of $y <_{\mathcal{S}} x$ where $<_{\mathcal{S}}$ is the precedence relation on the set of events of the model $\mathcal{S}$. So $\Omega(b) \not>_{\mathcal{S}} b$

just means that the reason for Poushka's barking does not come after his barking. Sometimes $\Omega(b) <_{\mathcal{S}} b$ holds, but there are circumstances where the reason for the barks continue even after the barks have stopped; for example, when Poushka recognizes that the person comming is in fact a friend. If we collect all the possible event occurrences that are reasons for a barking of Poushka into a set called R, then we expect that there is never $r \in R$ such that $\Omega(b) <_{\mathcal{S}} r <_{\mathcal{S}} b$.

A function with these two properties will be called *regular*. Regular functions appear naturally in the context of *regular registers* as defined by Lamport [1986]. A generalization of regular functions give *bounded functions*.

Definition 2.6 A partial function $\Omega\ : E_{\mathcal{S}} \to A \subseteq E_{\mathcal{S}}$ is called *bounded* of depth k, iff for all $x \in E_{\mathcal{S}}$, $\Omega(x) \not>_{\mathcal{S}} x$, and there is no sequence $x_1, \ldots, x_k \in A$ such that $\Omega(x) <_{\mathcal{S}} x_1 <_{\mathcal{S}} \ldots x_k <_{\mathcal{S}} x$. A bounded function of depth 1 is called a *regular* function.

In our experience unary functions of bounded depth suffice to model situations arizing in computing, and we suggest this class as a natural candidate for functions to have in system executions.

The finiteness axiom suggested by Lamport states that for each (terminating) event e the number of events x for which $e <_{\mathcal{S}} x$ does not occur is finite. Thus, intuitively, this axiom assumes a starting time and a fixed number of processes at each moment (this number may increase). For any process and moment there are only finitely many operation executions by that process that began before that moment, and hence the rationale for this axiom. It follows from this finiteness axiom that any antichain (*pairwise* $<_{\mathcal{S}}$ *incomparable set of events*) in $E_{\mathcal{S}}$ is finite. We tend to assume that there is even a fixed finite bound on the size of the antichains. An equivalent formulation that we adopt is that there is a fixed finite number of serial (linear) processes that constitute $E_{\mathcal{S}}$. (This equivalence is known as Dilworth's decomposition theorem; we need not use this theorem as we just assume the decomposition of $E_{\mathcal{S}}$ into finitely many chains.)

Finally, we make the distinction between 'initial', 'actual', and 'final' event occurrences in our structures. Take again the structures describing an afternoon in Poushka's life, and suppose that the starting time of the description is 12:00. The event occurrences that happen, or continue, after 12:00 are actual in this structure. If, for example, the first afternoon event is a barking, and its reason happened before 12:00, then something is missing from our model, which only contain the afternoon events, and we may have to introduce the cause of the bark as an initial (or 'prehistoric') event occurrence.

Once the need for initial events is recognized, the use of 'final' events becomes apparent too. In the structure describing the morning activities of Poushka, the event of someone approaching the fence is recorded as a final event. In general, the final events of a finite system execution $\mathcal{S}$ are those occurrences needed as initial events for any system execution that is following $\mathcal{S}$.

An event occurrence that is both in the 'morning' model and the 'afternoon' model is *unfinished* in the morning model, since it continues afterwards (it may finish of course in the afternoon model). In general, an unfinished event is one that we expect to continue to occur in any continuation of the system execution. We now have enough intuition for the formal definition given in the next section.

3 System executions

Define a *system execution* to be a structure

$$\mathcal{S} = (E_{\mathcal{S}}, M_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}})$$

where:

1. $E_{\mathcal{S}}$ is called the set of event occurrences; it is either finite or countably infinite.

2. $M_{\mathcal{S}}$ is the set of 'fixed objects'; it is assumed here to be finite. For notational simplicity, or in case it is empty, $M_{\mathcal{S}}$ may be omitted.

3. $<_{\mathcal{S}}$, the 'precedence relation', satisfies the following

   (a) $(E_{\mathcal{S}}, <_{\mathcal{S}})$ is an interval order;

   (b) Lamport's finiteness axiom holds. (A variant of the definition is to assume that $E_{\mathcal{S}}$ is union of finitely many chains, each is either finite or of the order type of the natural numbers.)

4. $F_{\mathcal{S}}$ is a finite collection of unary predicates defined on $T = E_{\mathcal{S}} \cup M_{\mathcal{S}}$, and partial unary functions from T to T. (A variant is to demand that the functions are of bounded depth—see Definition 2.6.) The predicates are partial since we deal with many sorted structures. A partial predicate has a domain and it defines a subset of that domain; outside its domain it is meaningless to ask if the predicate holds or not.

   In all of our applications, the functions are partial functions with the property that for any finite set X the closure of X under all the partial functions in $F_{\mathcal{S}}$ is finite, and is in fact obtained at once as the union of the ranges $f''X$ for $f \in F_{\mathcal{S}}$.

A system execution satisfying these four conditions is called a 'plain' system execution. The term system execution (or *full* system execution for emphasize) refers to one that is obtained by adding three special predicates on $E_{\mathcal{S}}$: I, A and T. They describe the three, not necessarily disjoint, subsets of $E_{\mathcal{S}}$: $I_{\mathcal{S}}$ which is called the 'initial' events, $A_{\mathcal{S}}$ the 'actual' events and $T_{\mathcal{S}}$ the 'final' events of $\mathcal{S}$. The sets $I_{\mathcal{S}}$ and $T_{\mathcal{S}}$ are finite, and $T_{\mathcal{S}} \subseteq I_{\mathcal{S}} \cup A_{\mathcal{S}}$. The initial events form an initial segment, and the actual events a final segment in $\mathcal{S}$, and their intersection is an antichain. So, if $a \in I_{\mathcal{S}}$ and $b \in A_{\mathcal{S}}$, then $a \not> _{\mathcal{S}} b$. If $a \in I_{\mathcal{S}} \setminus A_{\mathcal{S}}$ and $b \in A_{\mathcal{S}} \setminus I_{\mathcal{S}}$, then $a \ <_{\mathcal{S}} \ b$.

A special predicate in $F_{\mathcal{S}}$ is the *finished/unfinished* predicate (it may be present in plain system executions as well). In an infinite system execution, all the events are finished, but some events may be unfinished in a finite system execution. In a full system execution, the unfinished events of $E_{\mathcal{S}}$ are all in $T_{\mathcal{S}}$, but it is possible for $T_{\mathcal{S}}$ to also have finished events. The events of $I_{\mathcal{S}} \setminus A_{\mathcal{S}}$ are finished. If $a <_{\mathcal{S}} b$ then a is finished.

As an example, take the system execution that describes Poushka's afternoon. The actual events are those occurring after midday (including the ones that began before but continue till after midday). For example, $I_{\mathcal{S}}$ may include an event A of someone approaching the fence if it is used to explain an afternoon barking. In case A stretches over till after 12:00, then it is in $I_{\mathcal{S}} \cap A_{\mathcal{S}}$; it is unfinished and therefore a final event in the morning system execution.

Let us emphasize that the finished/unfinished distinction is not the same as the terminating/nonterminating distinction made by Lamport [1986(b)]. An event is nonterminating if its duration is infinite. In this paper, for simplicity, we only have terminating events. An unfinished event in the finite system execution $\mathcal{S}$ has a finite duration, but it has not yet been completed—it must be present in any continuation of $\mathcal{S}$.

Definition 3.1 A system execution $\mathcal{S}$ in which the actual occurrences $A_{\mathcal{S}}$ form a (possibly empty) antichain is called a *state*. If $A_{\mathcal{S}} = \emptyset$, then $\mathcal{S}$ is called an 'eventless' state.

An antichain has by definition no precedence relation on its members. Maximal antichains were called 'moments' in the proof of Theorem 2.3. A state is thus a system execution that describes a moment.

Two finite plain system executions are identified with each full system execution: its initial and final part. Let us first define the initial part of a system execution $\mathcal{S}$.

Given a full system execution $\mathcal{S} = (E_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}})$, the *initial part* of $\mathcal{S}$ is the plain system execution $\mathcal{U} = (E_{\mathcal{U}}, <_{\mathcal{U}}, F_{\mathcal{U}})$, defined by taking $E_{\mathcal{U}} = I_{\mathcal{S}}$ with $<_{\mathcal{U}}$ and $F'_{\mathcal{U}}$ being the restrictions of $<_{\mathcal{S}}$ and $F'_{\mathcal{S}}$ to $I_{\mathcal{S}}$, where $F'_{\mathcal{S}}$ denotes the collection $F_{\mathcal{S}}$ without the finished/unfinished and the $I_{\mathcal{S}},\ A_{\mathcal{S}},\ T_{\mathcal{S}}$ predicates. The unfinished events of $E_{\mathcal{U}}$ are defined to be $I_{\mathcal{S}} \cap A_{\mathcal{S}}$.

Denote with $F''_{\mathcal{S}}$ the collection $F_{\mathcal{S}}$ without the $I_{\mathcal{S}},\ A_{\mathcal{S}}$, and $T_{\mathcal{S}}$ predicates. The *final part* of $\mathcal{S}$ is the plain system execution defined by restricting $<_{\mathcal{S}}$ and $F''_{\mathcal{S}}$ to the final events $T_{\mathcal{S}}$. That is, $\mathcal{F} = (T_{\mathcal{S}}, (<_{\mathcal{S}} \lceil T_{\mathcal{S}}), F''_{\mathcal{S}} \lceil T_{\mathcal{S}})$. The unfinished events of $\mathcal{F}$ are thus the unfinished events of $\mathcal{S}$. The intuition for this definitions is that the final part of $\mathcal{S}$ constitutes the initial part of any continuation of $\mathcal{S}$.

In the next section we will see how to concatenate two (full) system executions $\mathcal{A}$ and $\mathcal{B}$ when the final part of $\mathcal{A}$ equals the initial part of $\mathcal{B}$. In particular, we will define the concatenation of a sequence of (full) states, $\mathcal{S}_i$, when the final part of $\mathcal{S}_i$ equals the initial part of $\mathcal{S}_{i+1}$. The final and initial parts of the states $\mathcal{S}_i$ are thus used as glue that binds the states into a full system execution.

Though it is natural to call the final part of a finite system execution its final *state*, we used the term final and initial *parts* to emphasize that these are different (though related) concepts. The final and initial parts are not seen here as a description of a moment, but rather as the necessary glue that allows the definition of concatenation. The final and initial parts are plain system executions and not states in the formal sense of Definition 3.1. To sharpen this distinction, suppose that some system execution $\mathcal{S}$ is presented as a concatenation of a sequence of states $\mathcal{S}_i,\ i \leq n$. Then $\mathcal{S}_n$, the last state in this decomposition, can be called a final state of $\mathcal{S}$. But there may be many such decompositions of $\mathcal{S}$ into a sequence of states, and thus different states may happen to be final states in such sequences which all concatenate to $\mathcal{S}$. On the other hand, the final part of $\mathcal{S}$ does not depend on a particular decomposition and is definable from $\mathcal{S}$ (it is also the final part of $\mathcal{S}_n$ and of any other final state of $\mathcal{S}$).

4 Operations on system executions: Concatenation and Concurrent unions

In this section the concatenation of two system executions and then their concurrent union is defined.

4.1 Concatenation

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

So, it is possible that the concatenation identifies some events in $A_{\mathcal{A}}$ (actual events of $\mathcal{A}$) with events in $A_{\mathcal{B}}$. This happens when an event in $A_{\mathcal{B}}$ is also an initial event in $I_{\mathcal{B}} \approx \mathcal{S}$, then it is unfinished in $\mathcal{S}$ and so it is unfinished in the occurrence of $\mathcal{S}$ as the final part of $\mathcal{A}$. Apart from these identifications, events of $\mathcal{A}$ precede the actual events of $\mathcal{B}$ in $\mathcal{A};\mathcal{B}$. It is not difficult to verify that the Russell-Wiener property holds in $\mathcal{A};\mathcal{B}$. We say that $\mathcal{A}$ is an initial (and $\mathcal{B}$ is a final) subexecution of $\mathcal{A};\mathcal{B}$.

In particular, it is possible now to define the concatenation of a sequence $\langle \mathcal{S}_i | i < i_0 \rangle$ (i_0 finite or ω) of states. It is assumed that for every i the final part of $\mathcal{S}_i$ equals the initial part of $\mathcal{S}_{i+1}$ in order for this concatenation to be defined.

A definition of the concept of a system can be given now. A *system* is a collection $\mathcal{S}$ of system executions (in a fixed language) such that if $\mathcal{S} \in S$ then any initial subexecution of $\mathcal{S}$ is in S, and if $\mathcal{A}_1$ and $\mathcal{A}_2$ are in S and have the same final parts, then the set of possible continuations of $\mathcal{A}_1$ and of $\mathcal{A}_2$ that are in S are the same. (That is, for any $\mathcal{B}$, $\mathcal{A}_1$; $\mathcal{B}$ is in S iff $\mathcal{A}_1$; $\mathcal{B}$ is in S.)

4.2 Concurrent Unions

The concurrent union of two system executions is the collection of system executions obtained by 'running' the two system executions concurrently.

It is convenient to use the notion of a *restriction* of a system execution in order to define the concurrent union of two system executions. For this definition, let us assume in our language a collection of special unary predicates which are called 'processes' names'. If P is such a process' name, then $P(a)$ is expressed by saying that a is in process P.

Restrictions: Let $\mathcal{S} = (E_\mathcal{S}, <_\mathcal{S}, F_\mathcal{S})$ be a system execution, and G be a set that consists of some processes' names and some relation and function symbols. The *restriction* of $\mathcal{S}$ to G is the system execution obtained by restricting those functions and predicates of $F_\mathcal{S}$ with names in G to the subset of $E_\mathcal{S}$ obtained by taking only the processes indicated in G. In model theoretic terms, a restriction is obtained by passing to the submodel containing only the indicated processes, and taking a *reduct* of this submodel by interprating only the symbols in G.

Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two system executions. Put $E_i = E_{\mathcal{S}_i}$ for $i = 1, 2$. And let L_i be the language interpreted by $F_{\mathcal{S}_i}$ (a set of processes' names, relation symbols, and function symbols.) Then $(\mathcal{S}_1 \| \mathcal{S}_2)$ is defined to be the system of all system executions $\mathcal{S}$ in the language $L_1 \cup L_2$ such that the restriction of $\mathcal{S}$ to L_i is $\mathcal{S}_i$ for $i = 1, 2$. (Actually, $\|_{L_1,L_2}$ is a more accurate notation.) Intuitively, $(\mathcal{S}_1 \| \mathcal{S}_2)$ is the collection of all system executions that can be simultaneously seen as consisting of two parts, one is $\mathcal{S}_1$ and the other is $\mathcal{S}_2$.

It is not difficult to see that $(\mathcal{S}_1 \| \mathcal{S}_2)$ is not empty if the restrictions of $\mathcal{S}_1$ and $\mathcal{S}_2$ to $L_1 \cap L_2$ are the same. Two cases are interesting here: the one in which $\mathcal{S}_1$ and $\mathcal{S}_2$ have disjoint processes, and the other in which one or more processes are identified in $\mathcal{S}_1$ and $\mathcal{S}_2$. The first case is when the names of the processes appearing in L_1 and in L_2 are disjoint, and the second case is when there are some common processes' names. This is the case, for example, when $\mathcal{S}_1$ describe one section and $\mathcal{S}_2$ another section in an orchestra. The common part is perhaps the conductor's instructions and tempo. Our definition is such that, in such a case, the system executions in $(\mathcal{S}_1 \| \mathcal{S}_2)$ identify and do not duplicate the common events.

4.3 Synchronizing processes

In some cases we may wish to define the concurrent union not as a system of all possible runs, but as the unique system execution resulting from the concurrent run. This is possible if the two systems share a common time scale and the union is made in accordance with it. To make this precise we need a definition. Let $\mathcal{S}$ be a system execution. A (serial) process $S \subset E_S$ is called a *synchronizing process* if the following holds:

1. Any antichain in $E_{\mathcal{S}}$ has a non empty intersection with S, and

2. every 'tick' $t \in S$ is contained in a *unique* antichain of $E_{\mathcal{S}}$.

Any system execution, $\mathcal{S}$, can be extended by adding a synchronizing process. A synchronizing process defines a representation of $\mathcal{S}$ and thus completely determines $<_{\mathcal{S}}$. This representation is into the intervals of the linear order formed by the synchronizing process: each event e is associated with those ticks t such that $\{e, t\}$ is an antichain.

If the synchronizing process of $\mathcal{S}_1$ is identified with that of $\mathcal{S}_2$, and the synchronizing process continues to be a synchronizing process for the concurrent union as well, then a single system execution results in $\mathcal{S}_1 \| \mathcal{S}_2$.

5 Cores of system executions

In this section we give a different interpretation to the relations $\longrightarrow$ and $\dashrightarrow$ defined and used by Lamport [1986] in his description of system executions. We also give here an alternative definition to the notion of a 'core' given by Gaifman [.].

The *atemporal* aspect of a system execution $\mathcal{S} = (E_{\mathcal{S}}, <_{\mathcal{S}}, F_{\mathcal{S}})$ is denoted $\log(\mathcal{S})$; it consists only of the record of the event occurrences and their properties. That is, $\log(\mathcal{S})$ is the structure $(E_{\mathcal{S}}, F_{\mathcal{S}})$ obtained by omitting $<_{\mathcal{S}}$ from $\mathcal{S}$.

Recall that a *system* is a collection of system executions.

Let Y be a system, and $\mathcal{S} \in Y$ be a system execution; the *core* of $\mathcal{S}$ (relative to Y) is the following structure $\mathcal{C} = (E_{\mathcal{C}}, \longrightarrow_{\mathcal{C}}, \dashrightarrow_{\mathcal{C}}, F_{\mathcal{C}})$.

(1) $E_\mathcal{C} = E_\mathcal{S}$, $F_\mathcal{C} = F_\mathcal{S}$. That is, $(E_\mathcal{C}, F_\mathcal{C}) = \log(\mathcal{S})$. For emphasize we may write $e_\mathcal{C}$ rather than e when the event $e \in E_\mathcal{S}$ is viewed as an element of $E_\mathcal{C}$.

(2) $a \longrightarrow_\mathcal{C} b$ iff $a <_\mathcal{R} b$ for all system executions $\mathcal{R} \in Y$ such that $\log(\mathcal{R}) = \log(\mathcal{S})$.

(3) $a \dashrightarrow_\mathcal{C} b$ iff for all $\mathcal{R} \in Y$ such that $\log(\mathcal{R}) = \log(\mathcal{S})$, $a \not>_\mathcal{R} b$ holds.

Intuitively, the core of $\mathcal{S}$ represents the common properties (temporal properties as well) of all the system executions in Y that share with $\mathcal{S}$ the same atemporal properties.

S. Ben David [1988] has shown the following properties of $\longrightarrow_\mathcal{C}$ and $\dashrightarrow_\mathcal{C}$.

(1) $\longrightarrow$ is an irreflexive partial order on E.

(2) $a \longrightarrow b$ implies $a \dashrightarrow b$ and $\neg(b \dashrightarrow a)$.

(3) $a \longrightarrow b \dashrightarrow c$ implies $a \dashrightarrow c$. Similarly, $a \dashrightarrow b \longrightarrow c$ implies $a \dashrightarrow c$.

(4) $a \longrightarrow b \dashrightarrow c \longrightarrow d$ implies $a \longrightarrow d$.

(5) $a \dashrightarrow b \longrightarrow c \dashrightarrow d$ implies $a \dashrightarrow d$ (Anger [1989]).

The finiteness property of Lamport does not necessarily hold in the core. But we have the following substitute

(6) 1. $\{y \in E \mid y \dashrightarrow x\}$ is finite.

2. E is the union of finitely many $\longrightarrow$ chains.

The global-time axiom is the statement

$$\forall x, y\ [x \dashrightarrow y \text{ iff } \neg(y \longrightarrow x)].$$

The global-time axiom needs not be true in the core $\mathcal{C}$.

The axioms above are identical to Lamport's [1986] axioms (except for the finiteness axiom) and we propose to offer the core of system executions as an explanation of Lamport's axioms. So in our view, these axioms represent the common temporal features of all runs of the system that have the same log.

We propose that the elements of $E_{\mathcal{C}}$ be called **events** (rather than event occurrences). The map that takes $e \in E_{\mathcal{S}}$ to $e = e_{\mathcal{C}} \in E_{\mathcal{C}}$, where $\mathcal{C}$ is the core of $\mathcal{S}$, associates to each occurrence its event type. (Of course, the equality $e = e_{\mathcal{C}}$ should not be interprated as saying that occurrences and events are the same. It is only by its context, $\mathcal{S}$ or $\mathcal{C}$, that a member of the structure acquires its meaning.) It seems reasonable to interpret $\longrightarrow$ as "necessarily precedes", and $a \dashrightarrow b$ as "causality".

Gaifman [.] defines *core* as a least constrained pomset in a system. If we do so, we would have to introduce *core*(S) as a system execution in the system Y. This entails relaxing the notion of a pomset by renouncing the global time axiom—something we don't find very intuitive but which is certainly possible.

Gaifman offered to define when two events are concurrent. In our terms, this amounts to the following. Let $a,\ b$ be two events in $\mathcal{C} = core(\mathcal{S})$. We say that a and b are concurrent if

$$\neg(a \dashrightarrow_{\mathcal{C}} b) \text{ and } \neg(b \dashrightarrow_{\mathcal{C}} a)$$

Equivalently: There are system executions $\mathcal{S}_1$ and $\mathcal{S}_2$ in Y, with the same log as $\log(\mathcal{S})$ such that $b <_{\mathcal{S}_1} a$ and $a <_{\mathcal{S}_2} b$.

Variations are possible. We suggest that if both $a \dashrightarrow b$ and $b \dashrightarrow a$, then a and b are "overlapping".

So far, in defining the events to be the elements of the core, two distinct occurrences $x,\ y \in E_{\mathcal{S}}$ have also distinct event types. Yet there are situations in which we would like to be able to declare that these x and y have the same event type. Let us indicate a partial solution to this problem. Given a system S and a system execution $\mathcal{S} \in S$, suppose $\mathcal{S}$ is the concatenation of a sequence $\mathcal{S}_i,\ i \in I$ of system executions in S (where the index set I can be finite or of the order type ω). Suppose further all the $\mathcal{S}_i$'s have the same core C. Then two occurrences $x, y \in E_{\mathcal{S}}$ (occurring in distinct $\mathcal{S}_i$ and $\mathcal{S}_j$) have the same event type if $x_{\mathcal{C}} = y_{\mathcal{C}}$. Similarly, $x_{\mathcal{C}} \longrightarrow_{\mathcal{C}} y_{\mathcal{C}}$ can be meaningfully claimed for $x, y \in E_{\mathcal{S}}$ even if $y <_{\mathcal{S}} x$ holds. Thus, for example, I can say that Poushka always cries and jumps for joy after I take the leash in my hand, even though I don't mean of course to say that yesterday's joy comes after today's holding of the leash.

6 Conclusion

We have seen here that the approaches to modeling and specification of systems taken by Lamport [1986] in defining system executions, and by Gischer [1984] and Pratt [1986] in defining pomsets, are not very far from each other. We hope that an accepted unified terminology will express this fact. Different authors have combined these two approaches. Gaifman [.] called *full* pomsets those pomsets with an interval order. Ben David [1988] introduces monadic predicates to Lamport's structures. It is quite clear that Lamport [1986] himself is informally dealing with richer structures than the system executions actually defined there. The structures developed here contain all these combined aspects, and more, while remaining relatively simple. They allow the use of fixed objects which are not event occurrences; the use of functions is more liberal, and the notions of concatenation and parallel executions allow the amalgamation of structures that have a common part.

Usually, a state is defined as a map that assigns values to the variables. Probst and Li [1990] have a different approach. They define a state as an equivalence class of pomsets that have the same possible continuations. (Two pomsets that have the same final parts are equivalent in this sense.) This is a valuable approach, but it makes a 'state' a second order concept. In our approach, a state is just a special kind of a pomset. Hence, whether one models reality as a sequence of states, or as a totality of events and relations, one remains within the same framework.

Let us point to some important issues that were not treated here. The specification of communication devices can be carried very naturally within the framework of system executions. As sample examples, see Gaifman's [.] discussion of a channel; or Lamport's [1986] characterization of atomic, regular and safe registers.

The importance of hierarchical decomposition of systems has been stressed by so many writers that we dare not point to a single reference; still, in our context, Lamport [1986] defines higher-level and lower-level views system executions which enable such a decomposition. Pratt [1986] discusses a similar concept. The facility to pass from one level to another within a single framework is a convincing demonstration in our opinion of the utility of the system execution concept.

To say that a sentence is true is to say that it holds in some structure, or a family of structures, and, similarly, to say that a program is correct is to claim a certain fact about structures that describe the program's executions. A formal correctness proof, being formal, does not mention any structures, yet definite models are clearly needed to express what proofs are about. The pomset concept seems to be very useful for this aim. Lamport [1979], [1986], and [1986(b)] contain correctness proofs of distributed protocols using system executions (the idea of a system execution, but not the term appears in the first paper). We hope to publish in subsequent work more examples of such proofs.

References

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

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

F. Anger

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

S. Ben-David

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

P. C. Fishburn

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

H. Gaifman

[.] *Modeling concurrency by partial orders and nonlinear transition systems*

J. Gischer

1984 *Partial Orders and the Axiomatic Theory of Shuffle* Ph.D Thesis, Computer Science Dept., Stanford University.

J. Gischer

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

L. Lamport
1979 *A New Approach to Proving the Correctness of Multiprocess Programs* ACM Transaction on Programming Languages and Systems, Vol. 1, pp. 84-97.

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

L. Lamport
1986(b) *The Mutual Exclusion Problem: Part I—A Theory of Interprocess Communication.* J. of the ACM, Vol. 33, pp. 313-326.

K. R. Poper
1959 *The Logic of Scientific Discovery* Basic Books, Inc New York.

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

D. K. Probst and H. F. Li
1990 *Modelling Reactive Hardware Processes Using Partial Orders.* Semantics for Concurrency, Proceedings of the International BCS-FACS Workshop, pp.324-343.

A. A. Robb
1921 *The absolute Relations of Time and Space.* Cambridge Univ. Press, 1921. Cited in:
The Concept of Space and Time, (The conical order of time-space), M. Capek (ed). D. Reidel. Boston 1976.

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