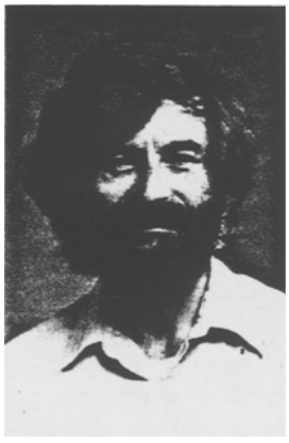
Distributed Computing (1986) 1:77-85

DISTRIBUTED COMPUTING

On interprocess communication

Part I: Basic formalism*

Leslie Lamport
Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA



Dr. Lamport *is a member of Digital Equipment Corporation's Systems Research Center. In previous incarnations, he was with SRI International and Massachusetts Computer Associates. The central topic of his research has been concurrency, and he can write* TEX *macros and chew gum at the same time.*

Abstract. A formalism for specifying and reasoning about concurrent systems is described. Unlike more conventional formalisms, it is not based upon atomic actions. A definition of what it means for one system to implement a higher-level system is given and justified. In Part II, the formalism is used to specify several classes of interprocess communication mechanisms and to prove the correctness of algorithms for implementing them.

Key words: Concurrent reading and writing – Nonatomic operations – Shared data

Introduction

This is the first part of a two-part paper addressing what I believe to be fundamental questions in the theory of interprocess communication. It develops a formal definition of what it means to implement one system with a lower-level one and provides a method for reasoning about concurrent systems. The definitions and axioms introduced here are applied in Part II [5] to algorithms that implement certain interprocess communication mechanisms.

To motivate the formalism, let us consider the question of atomicity. Most treatments of concurrent processing assume the existence of atomic operations – an atomic operation being one whose execution is performed as an indivisible action. The term *operation* is used to mean a class of actions such as depositing money in a bank account, and the term *operation execution* to mean one specific instance of executing such an action – for example, depositing \$100 in account number 14335 at 10:35 a.m. on December 14, 1987. Atomic operations must be implemented in terms of lower-level operations. A high-level language may provide a *P* operation to a semaphore as an atomic operation, but this operation must be implemented in terms of lower-level machine-language instructions. Viewed at the machine-language level, the semaphore operation is not atomic. Moreover, the machine-language operations must ultimately be implemented with circuits in which operations are manifestly nonatomic – the possibility of harmful "race conditions" shows that the setting and the testing of a flip-flop are not atomic actions.

Part II considers the problem of implementing atomic operations to a shared register with more primitive, nonatomic operations. Here, a more familiar example of implementing atomicity is used: concurrency control in a database. In a database system, higher-level transactions,

* Much of this research was performed while the author was a member of the Computer Science Laboratory at SRI International, where it was sponsored by the Office of Naval Research Project under contract number N00014-84-C-0621 and the Rome Air Development Command Project under contract number F30602-85-C-0024

which may read and modify many individual data items, are implemented with lower-level reads and writes of single items. These lower-level read and write operations are assumed to be atomic, and the problem is to make the higher-level transactions atomic. It is customary to say that a semaphore operation *is* atomic while a database transaction *appears to be* atomic, but this verbal distinction has no fundamental significance.

In database systems, atomicity of transactions is achieved by implementing a *serializable* execution order. The lower-level accesses performed by the different transactions are scheduled so that the net effect is the same as if the transactions had been executed in some serial order - first executing all the lower-level accesses comprising one transaction, then executing all the accesses of the next transaction, and so on. The transactions should not actually be scheduled in such a serial fashion, since this would be inefficient; it is necessary only that the effect be the same as if that were done.[1]

In the literature on concurrency control in databases, serializability is usually the only correctness condition that is stated [1]. However, serializability by itself does not ensure correctness. Consider a database system in which each transaction either reads from or writes to the database, but does not do both. Moreover, assume that the system has a finite lifetime, at the end of which it is to be scrapped. Serializability is achieved by an implementation in which reads always return the initial value of the database entries and writes are simply not executed. This yields the same results as a serial execution in which one first performs all the read transactions and then all the writes. While such an implementation satisfies the requirement of serializability, no one would consider it to be correct.

This example illustrates the need for a careful examination of what it means for one system to implement another. It is reconsidered in Sect. 2, where the additional correctness condition needed to rule out this absurd implementation is stated.

1 In the context of databases, atomicity often denotes the additional property that a failure cannot leave the database in a state reflecting a partially completed transaction. In this paper, the possibility of failure is ignored, so no distinction between atomicity and serializability is made

1 System executions

Almost all models of concurrent processes have indivisible atomic actions as primitive elements. For example, models in which a process is represented by a sequence or "trace" [10, 12, 13] assume that each element in the sequence represents an indivisible action. Net models [2] and related formalisms [9, 11] assume that the firing of an individual transition is atomic. These models are not appropriate for studying such fundamental questions as what it means to implement an atomic operation, in which the nonatomicity of operations must be directly addressed.

More conventional formalisms are therefore eschewed in favor of one introduced in [4] and refined in [3], in which the primitive elements are *operation executions* that are not assumed to be atomic. This formalism is described below; the reader is referred to [4] and [3] for more details.

A *system execution* consists of a set of *operation executions*, together with certain temporal precedence relations on these operation executions. Recall that an operation execution represents a single execution of some operation. When all operations are assumed to be atomic, an operation execution A can influence another operation execution B only if A precedes B - meaning that all actions of A are completed before any action of B is begun. In this case, one needs only a single temporal relation $\longrightarrow$, read "precedes", to describe the temporal ordering among operation executions. While temporal precedence is usually considered to be a total ordering of atomic operations, in distributed systems it is best thought of as an irreflexive partial ordering (see [6]).

Nonatomicity introduces the possibility that an operation execution A can influence an operation execution B without preceding it; it is necessary only that some action of A precede some action of B. Hence, in addition to the precedence relation $\longrightarrow$, one needs an additional relation $\dashrightarrow$, read "can affect", where $A \dashrightarrow B$ means that some action of A precedes some action of B.

Definition 1. A *system execution* is a triple $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$, where $\mathcal{S}$ is a finite or countably infinite set whose elements are called *operation executions*, and $\longrightarrow$ and $\dashrightarrow$ are precedence relations on $\mathcal{S}$ satisfying axioms A1-A5 below.

To assist in understanding the axioms for the $\longrightarrow$ and $\dashrightarrow$ relations, it is helpful to have a semantic model for the formalism. The model to be used is one in which an operation execution is represented by a set of primitive actions or events, where $A \longrightarrow B$ means that all the events of A precede all the events of B, and $A \dashrightarrow B$ means that some event of A precedes some event of B. Letting $\mathbf{E}$ denote the set of all events, and $\longrightarrow$ the temporal precedence relation among events, we get the following formal definition.

Definition 2. A *model* of a system execution $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ consists of a triple $\mathbf{E}, \longrightarrow, \mu$, where $\mathbf{E}$ is a set, $\longrightarrow$ is an irreflexive partial ordering on $\mathbf{E}$, and μ is a mapping that assigns to each operation execution A of $\mathscr{S}$ a nonempty subset $\mu(A)$ of $\mathbf{E}$, such that for every pair of operation executions A and B of $\mathscr{S}$:

$$A \longrightarrow B \equiv \forall\, a \in \mu(A): \forall\, b \in \mu(B): a \longrightarrow b$$
$$A \dashrightarrow B \equiv \exists\, a \in \mu(A): \exists\, b \in \mu(B): a \longrightarrow b \text{ or } a = b. \quad (1)$$

Note that the same symbol $\longrightarrow$ denotes the "precedes" relation both between operation executions in $\mathscr{S}$ and between events in $\mathbf{E}$.

Other than the existence of the temporal partial-ordering relation $\longrightarrow$, no assumption is made about the structure of the set of events $\mathbf{E}$. In particular, operation executions may be modeled as infinite sets of events. An important class of models is obtained by letting $\mathbf{E}$ be the set of events in four-dimensional spacetime, with $\longrightarrow$ the "happens before" relation of special relativity, where $a \longrightarrow b$ means that it is temporally possible for event a to causally affect event b.

Another simple and useful class of models is obtained by letting $\mathbf{E}$ be the real number line and representing each operation execution A as a closed interval.

Definition 3. A *global-time model* of a system execution $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ is one in which $\mathbf{E}$ is the set of real numbers, $\longrightarrow$ is the ordinary $<$ relation, and each set $\mu(A)$ is of the form $[s_A, f_A]$ with $s_A < f_A$.

Think of s_A and f_A as the starting and finishing times of A. In a global-time model, $A \longrightarrow B$ means that A finishes before B starts, and $A \dashrightarrow B$ means that A starts before (or at the same time as) B finishes. These relations are illustrated by Fig. 1, where operation executions

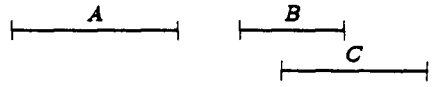


Fig. 1. Three operation executions in a global-time model

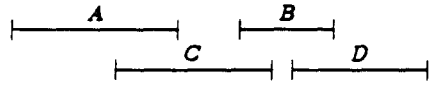


Fig. 2. An illustration of Axiom A4

A, B, and C, represented by the three indicated intervals, satisfy: $A \longrightarrow B$, $A \longrightarrow C$, $B \dashrightarrow C$, and $C \dashrightarrow B$. (In this and similar figures, the number line runs from left to right, and overlapping intervals are drawn one above the other.)

To complete Definition 1, the axioms for the precedence relations $\longrightarrow$ and $\dashrightarrow$ of a system execution must be given. They are the following, where A, B, C, and D denote arbitrary operation executions in $\mathscr{S}$. Axiom A4 is illustrated (in a global-time model) by Fig. 2; the reader is urged to draw similar pictures to help understand the other axioms.

A1. The relation $\longrightarrow$ is an irreflexive partial ordering.

A2. If $A \longrightarrow B$ then $A \dashrightarrow B$ and $B \not\dashrightarrow A$.

A3. If $A \longrightarrow B \dashrightarrow C$ or $A \dashrightarrow B \longrightarrow C$ then $A \dashrightarrow C$.

A4. If $A \longrightarrow B \dashrightarrow C \longrightarrow D$ then $A \longrightarrow D$.

A5. For any A, the set of all B such that $A \not\dashrightarrow B$ is finite.

(These axioms differ from the ones in [3] because only terminating operation executions are considered here.)

Axioms A1–A4 follow from (1), so they do not constrain the choice of a model. Axiom A5 does not follow from (1); it restricts the class of allowed models. Intuitively, A5 asserts that a system execution begins at some point in time, rather than extending into the infinite past. When $\mathbf{E}$ is the set of events in space-time, A5 holds for any model in which: (i) each operation occupies a finite region of space-time, (ii) any finite region of space-time contains only a finite number of operation executions, and (iii) the system is not expanding faster than the speed of light.[2]

Most readers will find it easiest to think about

2 A system expanding faster than the speed of light could have an infinite number of operation executions none of which are preceded by any operation

system executions in terms of a global-time model, and to interpret the relations $\longrightarrow$ and $\dashrightarrow$ as indicated by the example in Fig. 1. Such a mental model is adequate for most purposes. However, the reader should be aware that in a system execution having a global-time model, for any distinct operation executions A and B, either $A \longrightarrow B$ or $B \dashrightarrow A$. (In fact, this is a necessary and sufficient condition for a system execution to have a global-time model [8].) However, in a system execution without a global-time model, it is possible for neither $A \longrightarrow B$ nor $B \dashrightarrow A$ to hold. As a trivial counterexample, let $\mathscr{S}$ consist of two elements and let the relations $\longrightarrow$ and $\dashrightarrow$ be empty.

While a global-time model is a valuable aid to acquiring an intuitive understanding of a system, it is better to use more abstract reasoning when proving properties of systems. The relations $\longrightarrow$ and $\dashrightarrow$ capture the essential temporal properties of a system execution, and A1–A5 provide the necessary tools for reasoning about these relations. It has been my experience that proofs based upon these axioms are simpler and more instructive than ones that involve modeling operation executions as sets of events.

2 Hierarchical views

A system can be viewed at different levels of detail, with different operation executions at each level. Viewed at the customer's level, a banking system has operation executions such as *deposit* \$1000. Viewed at the programmer's level, this same system executes operations such as $dep\_amt[cust] := 1000$. The fundamental problem of system building is to implement one system (like a banking system) as a higher-level view of another system (like a Pascal program).

A higher-level operation consists of a set of lower-level operations – the set of operations that implement it. Let $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ be a system execution and let $\mathscr{H}$ be a set whose elements, called *higher-level operation executions*, are sets of operation executions from $\mathscr{S}$. A model for $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ represents each operation execution in $\mathscr{S}$ by a set of events. This gives a representation of each higher-level operation execution H in $\mathscr{H}$ as a set of events – namely, the set of all events contained in the representation of the lower-level operation executions that comprise H. This in turn defines precedence relations $\xrightarrow{*}$ and $\dashrightarrow^{*}$, where $G \xrightarrow{*} H$ means that all events in (the representation of) G precede all events in H, and $G \dashrightarrow^{*} H$ means that some event in G precedes some event in H, for G and H in $\mathscr{H}$.

To express all this formally, let $\mathbf{E}$, $\longrightarrow$, μ be a model for $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$, define the mapping μ^* on $\mathscr{H}$ by

$$\mu^*(H) = \bigcup \{\mu(A) : A \in H\}$$

and define the precedence relations $\xrightarrow{*}$ and $\dashrightarrow^{*}$ on $\mathscr{H}$ by

$$G \xrightarrow{*} H \equiv \forall\, g \in \mu^*(G) : \forall\, h \in \mu^*(H) : g \longrightarrow h$$
$$G \dashrightarrow^{*} H \equiv \exists\, g \in \mu^*(G) : \exists\, h \in \mu^*(H) : g \longrightarrow h \text{ or } g = h.$$

Using (1), it is easy to show that these precedence relations are the same ones obtained by the following definitions:

$$G \xrightarrow{*} H \equiv \forall\, A \in G : \forall\, B \in H : A \longrightarrow B$$
$$G \dashrightarrow^{*} H \equiv \exists\, A \in G : \exists\, B \in H : A \dashrightarrow B \text{ or } A = B. \quad (2)$$

Observe that $\xrightarrow{*}$ and $\dashrightarrow^{*}$ are expressed directly in terms of the $\longrightarrow$ and $\dashrightarrow$ relations on $\mathscr{S}$, without reference to any model. We take (2) to be the definition of the relations $\xrightarrow{*}$ and $\dashrightarrow^{*}$.

For the triple $\langle \mathscr{H}, \xrightarrow{*}, \dashrightarrow^{*} \rangle$ to be a system execution, the relations $\xrightarrow{*}$ and $\dashrightarrow^{*}$ must satisfy axioms A1–A5. If each element of $\mathscr{H}$ is assumed to be a nonempty set of operation executions, then Axioms A1–A4 follow from (2) and the corresponding axioms for $\longrightarrow$ and $\dashrightarrow$. For A5 to hold, it is sufficient that each element of $\mathscr{H}$ consist of a finite number of elements of $\mathscr{S}$, and that each element of $\mathscr{S}$ belong to a finite number of elements of $\mathscr{H}$. Adding the natural requirement that every lower-level operation execution be part of some higher-level one, this leads to the following definition.

Definition 4. A *higher-level view* of a system execution $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ consists of a set $\mathscr{H}$ such that:

H1. Each element of $\mathscr{H}$ is a finite, nonempty set of elements of $\mathscr{S}$.
H2. Each element of $\mathscr{S}$ belongs to a finite, nonzero number of elements of $\mathscr{H}$.

In most cases of interest, $\mathscr{H}$ is a partition of $\mathscr{S}$, so each element of $\mathscr{S}$ belongs to exactly one element of $\mathscr{H}$. However, Definition 4 allows the more general case in which a single lower-level operation execution is viewed as part of the implementation of more than one higher-level one.

Let us now consider what it should mean

for one system to implement another. If the system execution $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ is an implementation of a system execution $\langle \mathscr{H}, \xrightarrow{\mathscr{H}}, \dashrightarrow_{\mathscr{H}} \rangle$, then we expect $\mathscr{H}$ to be a higher-level view of $\mathscr{S}$ - that is, each operation in $\mathscr{H}$ should consist of a set of operation executions of $\mathscr{S}$ satisfying H1 and H2. This describes the elements of $\mathscr{H}$, but not the precedence relations $\xrightarrow{\mathscr{H}}$ and $\dashrightarrow_{\mathscr{H}}$. What should those relations be?

If we consider the operation executions in $\mathscr{S}$ to be the "real" ones, and the elements of $\mathscr{H}$ to be fictitious groupings of the real operation executions into abstract, higher-level ones, then the induced precedence relations $\xrightarrow{*}$ and $\dashrightarrow^{*}$ represent the "real" temporal relations on $\mathscr{H}$. These induced relations make the higher-level view $\mathscr{H}$ a system execution, so they are an obvious choice for the relations $\xrightarrow{\mathscr{H}}$ and $\dashrightarrow_{\mathscr{H}}$. However, as we shall see, they may not be the proper choice.

Let us return to the problem of implementing atomic database operations. Atomicity requires that, when viewed at the level at which the operation executions are the transactions, the transactions appear to be executed sequentially. In terms of our formalism, the correctness condition is that, in any system execution $\langle \mathscr{H}, \xrightarrow{\mathscr{H}}, \dashrightarrow_{\mathscr{H}} \rangle$ of the database system, all the elements of $\mathscr{H}$ (the transactions) must be totally ordered by $\xrightarrow{\mathscr{H}}$. This higher-level view of the database operations is implemented by lower-level operations that access individual database items. The higher-level system execution $\langle \mathscr{H}, \xrightarrow{\mathscr{H}}, \dashrightarrow_{\mathscr{H}} \rangle$ must be implemented by a lower-level one $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ in which each transaction H in $\mathscr{H}$ is implemented by a set of lower-level operation executions in $\mathscr{S}$.

Suppose $G = \{G_1, \ldots, G_m\}$ and $H = \{H_1, \ldots, H_n\}$ are elements of $\mathscr{H}$, where the G_i and H_i are operation executions in $\mathscr{S}$. For $G \xrightarrow{*} H$ to hold, each G_i must precede ($\longrightarrow$) each H_j, and, conversely, $H \xrightarrow{*} G$ only if each H_j precedes each G_i. In a situation like the one in Fig. 3, neither $G \xrightarrow{*} H$ nor $H \xrightarrow{*} G$ holds. (For a system with a global-time model, this means that both $G \dashrightarrow^{*} H$ and $H \dashrightarrow^{*} G$ hold.) If we required that the relations $\xrightarrow{\mathscr{H}}$ and $\dashrightarrow_{\mathscr{H}}$ be the induced relations $\xrightarrow{*}$ and $\dashrightarrow^{*}$, then the only way to implement a serializable system, in which $\xrightarrow{\mathscr{H}}$ is a total ordering of the transactions, would be to prevent the type of interleaved execution shown in Fig. 3. The only allowable system executions would be those in which the transactions were actually executed serially - each transaction being completed before the next one is begun.

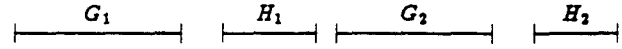


Fig. 3. An example with $G \longrightarrow H$ and $H \longrightarrow G$

Serial execution is, of course, too stringent a requirement because it prevents the concurrent execution of different transactions. We merely want to require that the system behave *as if* there were a serial execution. To show that a given system correctly implements a serializable database system, one specifies both the set of lower-level operation executions corresponding to each higher-level transaction and the precedence relation $\xrightarrow{\mathscr{H}}$ that describes the "as if" order, where the transactions act as if they had occurred in that order. This order must be consistent with the values read from the database - each read obtaining the value written by the most recent write of that item, where "most recent" is defined by $\xrightarrow{\mathscr{H}}$.

As was observed in the introduction, the condition that a read obtain a value consistent with the ordering of the operations is not the only condition that must be placed upon $\xrightarrow{\mathscr{H}}$. For the example in which each transaction either reads from or writes to the database, but does not do both, we must rule out an implementation that throws writes away and lets a read return the initial values of the database entries - an implementation that achieves serializability with a precedence relation $\xrightarrow{\mathscr{H}}$ in which all the read transactions precede all the write transactions. Although this implementation satisfies the requirement that every read obtain the most recently written value, this precedence relation is absurd because a read is defined to precede a write that may really have occurred years earlier.

Why is such a precedence relation absurd? In a real system, these database transactions may occur deep within the computer; we never actually see them happen. What is wrong with defining the precedence relation $\xrightarrow{\mathscr{H}}$ to pretend that these operation executions happened in any order we wish? After all, we are already pretending, contrary to fact, that the operations occur in some serial order.

In addition to reads and writes to database items, real systems perform externally observable operation executions such as printing on terminals. By observing these operation executions, we can infer precedence relations among the internal reads and writes. We need some condition on $\xrightarrow{\mathscr{H}}$ and $\dashrightarrow_{\mathscr{H}}$ to rule out pre-

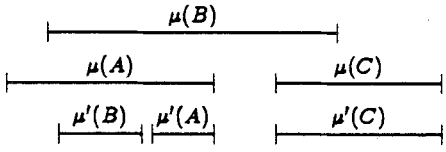


Fig. 4. An illustration of Proposition 1

cedence relations that contradict such observations.

It is shown below that these contradictions are avoided by requiring that if one higher-level operation execution "really" precedes another, then that precedence must appear in the "pretend" relations. Remembering that $\xrightarrow{*}$ and $\dashrightarrow^{*}$ are the "real" precedence relations and $\xrightarrow{\mathscr{H}}$ and $\dashrightarrow^{\mathscr{H}}$ are the "pretend" ones, this leads to the following definition.

Definition 5. A *system execution* $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ *implements* a system execution $\langle \mathscr{H}, \xrightarrow{\mathscr{H}}, \dashrightarrow^{\mathscr{H}} \rangle$ if $\mathscr{H}$ is a higher-level view of $\mathscr{S}$ and the following condition holds:

H3. For any $G, H \in \mathscr{H}$: if $G \xrightarrow{*} H$ then $G \xrightarrow{\mathscr{H}} H$, where $\xrightarrow{*}$ is defined by (2).

One justification for this definition in terms of global-time models is given by the following proposition, which is proved in [8]. (Recall that a global-time model is determined by the mapping μ, since the set of events and their ordering is fixed.)

Proposition 1. *Let* $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ *and* $\langle \mathscr{S}, \longrightarrow', \dashrightarrow' \rangle$ *be system executions, both of which have global-time models, such that for any* $A, B \in \mathscr{S}$: $A \longrightarrow B$ *implies* $A \longrightarrow' B$. *For any global-time model* μ *of* $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ *there exists a global-time model* μ' *of* $\langle \mathscr{S}, \longrightarrow', \dashrightarrow' \rangle$ *such that* $\mu'(A) \subseteq \mu(A)$ *for every* A *in* $\mathscr{S}$.

This proposition is illustrated in Fig. 4, where: (i) $\mathscr{S} = \{A, B, C\}$, (ii) A — C is the only $\longrightarrow$ relation, and (iii) $B \longrightarrow' A \longrightarrow' C$. To apply Proposition 1 to Definition 5, substitute $\mathscr{S}$ for $\mathscr{H}$, substitute $\xrightarrow{*}$ and $\dashrightarrow^{*}$ for $\longrightarrow$ and $\dashrightarrow$, and substitute $\xrightarrow{\mathscr{H}}$ and $\dashrightarrow^{\mathscr{H}}$ for $\longrightarrow'$ and $\dashrightarrow'$. The proposition then states that the "pretend" precedence relations are obtained from the real ones by shrinking the time interval during which the operation execution is considered to have occurred.

Let us return to the example of implementing a serializable database system. The formal requirement is that any system execution $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$, whose operation executions consist of reads and writes of individual database items, must implement a system $\langle \mathscr{H}, \xrightarrow{\mathscr{H}}, \dashrightarrow^{\mathscr{H}} \rangle$, whose operations are database transactions, such that $\xrightarrow{\mathscr{H}}$ is a total ordering of $\mathscr{H}$. By Proposition 1, this means that not only must the transactions be performed as if they had been executed in some sequential order, but that this order must be one that could have been obtained by executing each transaction within some interval of time during the period when it actually was executed. This rules out the absurd implementation described above, which implies a precedence relation $\xrightarrow{\mathscr{H}}$ that makes writes come long after they actually occurred.

Another justification for Definition 5 is derived from the following result, which is proved in [8]. Its statement relies upon the obvious fact that if $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ is a system execution, then $\langle \mathscr{T}, \longrightarrow, \dashrightarrow \rangle$ is also a system execution for any subset $\mathscr{T}$ of $\mathscr{S}$. (The symbols $\longrightarrow$ and $\dashrightarrow$ denote both the relations on $\mathscr{S}$ and their restrictions to $\mathscr{T}$. Also, in the proposition, the set $\mathscr{T}$ is identified with the set of all singleton sets $\{A\}$ for $A \in \mathscr{T}$.)

Proposition 2. *Let* $\mathscr{S} \cup \mathscr{T}, \longrightarrow, \dashrightarrow$ *be a system execution, where* $\mathscr{S}$ *and* $\mathscr{T}$ *are disjoint; let* $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ *be an implementation of a system execution* $\langle \mathscr{H}, \xrightarrow{\mathscr{H}}, \dashrightarrow^{\mathscr{H}} \rangle$; *and let* $\xrightarrow{*}$ *and* $\dashrightarrow^{*}$ *be the relations defined on* $\mathscr{H} \cup \mathscr{T}$ *by* (2). *Then there exist precedence relations* $\xrightarrow{\mathscr{H}\mathscr{T}}$ *and* $\dashrightarrow^{\mathscr{H}\mathscr{T}}$ *such that:*

- $\mathscr{H} \cup \mathscr{T}, \xrightarrow{\mathscr{H}\mathscr{T}}, \dashrightarrow^{\mathscr{H}\mathscr{T}}$ *is a system execution that is implemented by* $\mathscr{S} \cup \mathscr{T}, \longrightarrow, \dashrightarrow$.
- *The restrictions of* $\xrightarrow{\mathscr{H}\mathscr{T}}$ *and* $\dashrightarrow^{\mathscr{H}\mathscr{T}}$ *to* $\mathscr{H}$ *equal* $\xrightarrow{\mathscr{H}}$ *and* $\dashrightarrow^{\mathscr{H}}$, *respectively.*
- *The restrictions of* $\xrightarrow{\mathscr{H}\mathscr{T}}$ *and* $\dashrightarrow^{\mathscr{H}\mathscr{T}}$ *to* $\mathscr{T}$ *are extensions of the relations* $\xrightarrow{*}$ *and* $\dashrightarrow^{*}$, *respectively.*

To illustrate the significance of this proposition for Definition 5, let $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ be a system execution of reads and writes to database items that implements a higher-level system execution $\langle \mathscr{H}, \xrightarrow{\mathscr{H}}, \dashrightarrow^{\mathscr{H}} \rangle$ of database transactions. The operation executions of $\mathscr{S}$ presumably occur deep inside the computer and are not directly observable. Let $\mathscr{T}$ be the set of all other operation executions in the system, including the externally observable ones. Proposition 2 means that, while the "pretend" precedence relations $\xrightarrow{\mathscr{H}}$ and $\dashrightarrow^{\mathscr{H}}$ may imply new precedence relations on the operation exe-

cutions in $\mathscr{T}$, these relations ($\overset{\mathscr{H}\mathscr{T}}{\longrightarrow}$ and $\overset{\mathscr{H}\mathscr{T}}{\dashrightarrow}$) are consistent with the "real" precedence relations $\overset{\mathscr{T}}{\longrightarrow}$ and $\overset{\mathscr{T}}{\dashrightarrow}$ on $\mathscr{T}$. Thus, pretending that the database transactions occur in the order given by $\overset{\mathscr{H}}{\longrightarrow}$ does not contradict any of the real, externally observable orderings among the operations in $\mathscr{T}$.

When implementing a higher-level system, one usually ignores all operation executions that are not part of the implementation. For example, when implementing a database system, one considers only the transactions that access the database, ignoring the operation executions that initiate the transactions and use their results. This is justified by Proposition 2, which shows that the implementation cannot lead to any anomalous precedence relations among the operation executions that are being ignored.

A particularly simple kind of implementation is one in which each higher-level operation execution is implemented by a single lower-level one.

Definition 6. An implementation $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ of $\langle \mathscr{H}, \overset{\mathscr{H}}{\longrightarrow}, \overset{\mathscr{H}}{\dashrightarrow} \rangle$ is said to be *trivial* if every element of $\mathscr{H}$ is a singleton set.

In a trivial implementation, the sets $\mathscr{S}$ and $\mathscr{H}$ are (essentially) the same; the two system executions differ only in their precedence relations. A trivial implementation is one that is not an implementation in the ordinary sense, but merely involves choosing new precedence relations ("as if" temporal relations).

3 Systems

A system execution has been defined, but not a system. Formally, a system is just a set of system executions - a set that represents all possible executions of the system.

Definition 7. A *system* is a set of system executions.

The usual method of describing a system is with a program written in some programming language. Each execution of such a program describes a system execution, and the program represents the system consisting of the set of all such executions. When considering communication and synchronization properties of concurrent systems, the only operation executions that are of interest are ones that involve interprocess communication - for example, the operations of sending a message or reading a shared variable. Internal "calculation" steps can be ignored. If x, y, and z are shared variables and a is local to the process in question, then an execution of the statement $x := y + a * z$ includes three operation executions of interest: a read of y, a read of z, and a write of x. The actions of reading a, computing the product, and computing the sum are independent of the actions of other processes and could be considered to be either separate operation executions or part of the operation that writes the new value of x. For analyzing the interaction among processes, what is significant is that each of the two reads precedes ($\longrightarrow$) the write, and that no precedence relation is assumed between the two reads (assuming that the programming language does not specify an evaluation order within expressions).

A formal semantics for a programming language can be given by defining, for each syntactically correct program, the set of all possible executions. This is done by recursively defining a succession of lower and lower higher-level views, in which each operation execution represents a single execution of a syntactic program unit.[3] At the highest-level view, a system execution consists of a single operation execution that represents an execution of the entire program. A view in which an execution of the statement $S; T$ is a single operation execution is refined into one in which an execution consists of an execution of S followed by ($\longrightarrow$) an execution of T.[4] While this kind of formal semantics may be useful in studying subtle programming language issues, it is unnecessary for the simple language constructs generally used in describing synchronization algorithms like the ones in Part II, so these ideas will just be employed informally.

Having defined what a system is, the next step is to define what it means for a system **S** to implement a higher-level system **H**. The higher-level system **H** can be regarded as a specification of the lower-level one **S**, so we must decide what it should mean for a system to meet a specification.

The system executions of **S** involve lower-level concepts such as program variables; those

3 For nonterminating programs, the formalism must be extended to allow nonterminating higher-level operation executions, each one consisting of an infinite set of lower-level operation executions

4 In the general case, we must also allow the possibility that an execution of $S; T$ consists of a nonterminating execution of S

of **H** involve higher-level concepts such as transactions. The first thing we need is some way of interpreting a "concrete" system execution $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ of the "real" implementation **S** as an "abstract" execution of the "imaginary" high-level system **H**. Thus, there must be some mapping ι that assigns to any system execution $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ of **S** a higher-level system execution $\iota(\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle)$ that it implements. The implementation **S**, which is a set of system executions, yields a set $\iota(\mathbf{S})$ of higher-level system executions. What should be the relation between $\iota(\mathbf{S})$ and **H**?

There are two distinct approaches to specification, which may be called the *prescriptive* and *restrictive* approaches. The prescriptive approach is generally employed by methods in which a system is specified with a high-level program, as in [9] and [11]. An implementation must be equivalent to the specification in the sense that it exhibits all the same possible behaviors as the specification. In the prescriptive approach, one requires that every possible execution of the specification **H** be represented by some execution of **S**, so $\iota(\mathbf{S})$ must equal **H**.

The restrictive approach is employed primarily by axiomatic methods, in which a system is specified by stating the properties it must satisfy. Any implementation that satisfies those properties is acceptable; it is not necessary for the implementation to allow all possible behaviors that satisfy the properties. If **H** is the set of all system executions satisfying the required properties, then the restrictive approach requires only that every execution of **S** represent some execution of **H**, so $\iota(\mathbf{S})$ must be contained in **H**.

To illustrate the difference between the two approaches, consider the problem of implementing a program containing the statement $x := y + a * z$ with a lower-level machine-language program. The statement does not specify in which order y and z are to be read, so **H** should contain executions in which y is read before z, executions in which z is read before y, as well as ones in which they are read concurrently. With the prescriptive approach, a correct implementation would have to allow all of these possibilities, so a machine-language program that always reads y first then z would not be a correct implementation. In the restrictive approach, this is a perfectly acceptable implementation because it exhibits one of the allowed possibilities.

The usual reason for not specifying the order of evaluation is to allow the compiler to choose any convenient order, not to require that it produce nondeterministic object code. I therefore find the restrictive approach to be the more natural and adopt it in the following definition.

Definition 8. The system **S** *implements* a system **H** if there is a mapping $\iota: \mathbf{S} \mapsto \mathbf{H}$ such that, for every system execution $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ in **S**, $\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle$ implements $\iota(\langle \mathscr{S}, \longrightarrow, \dashrightarrow \rangle)$.

In taking the restrictive approach, one faces the question of how to specify that the system must actually do anything. The specification of a banking system must allow a possible system execution in which no customers happen to use an automatic teller machine on a particular afternoon, and it must include the possibility that a customer will enter an invalid request. How can we rule out an implementation in which the machine simply ignores all customer requests during an afternoon, or interprets any request as an invalid one?

The answer lies in the concept of an *interface specification*, discussed in [7]. The specification must explicitly describe how certain interface operations are to be implemented; their implementation is not left to the implementer. The interface specification for the bank includes a description of what sequences of keystrokes at the teller machine constitute valid requests, and the set of system executions only includes ones in which every valid request is serviced. What it means for someone to use the machine is part of the interface specification, so the possibility of no one using the machine on some afternoon does not allow the implementation to ignore someone who does use it.

Part II considers only the internal operations that effect communication between processes within the system, not the interface operations that effect communication between the system and its environment. Therefore, the interface specification is not considered further. The reader is referred to [7] for a discussion of this subject.

References

1. Bernstein PA, Goodman N (1981) Concurrency control in distributed database systems. ACM Comput Surv 13:185–222
2. Brauer W (ed) (1980) Net Theory and Applications. Lect Notes Comput Sci 84, Springer-Verlag, Berlin Heidelberg New York

3. Lamport L (in press) The mutual exclusion problem. J ACM
4. Lamport L (1979) A new approach to proving the correctness of multiprocess programs. ACM Trans Program Lang Syst 1:84–97
5. Lamport L. On interprocess communication. Part II: Algorithms. Distributed Computing 1:85–101
6. Lamport L (1978) Time, clocks and the ordering of events in a distributed system. Commun ACM 21:558–565
7. Lamport L (1985) What it means for a concurrent program to satisfy a specification: why no one has specified priority. In: Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN, New Orleans
8. Lamport L (1985) Interprocess Communication. SRI Technical Report, March 1985
9. Lauer PE, Shields MW, Best E (1979) Formal Theory of the Basic COSY Notation. Technical Report TR143, Computing Laboratory, University of Newcastle upon Tyne
10. Mazurkiewicz A (1984) Semantics of Concurrent Systems: A Modular Fixed Point Trace Approach. Technical Report 84–19, Institute of Applied Mathematics and Computer Science, University of Leiden
11. Milner R (1980) A Calculus of Communicating Systems. Lect Notes Comput Sci 92, Springer-Verlag, Berlin Heidelberg New York
12. Pnueli A (1977) The temporal logic of programs. In: Proc. of the 18th Symposium on the Foundations of Computer Science, ACM, November 1977
13. Winskel G (1980) Events in Computation. PhD thesis, Edinburgh University