On Interprocess Communication

Leslie Lamport

December 25, 1985

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.

Publication History

The two parts of this report will appear as separate articles in *Distributed Computing*.

Author's Abstract

A formalism, not based upon atomic actions, for specifying and reasoning about concurrent systems is defined. It is used to specify several classes of interprocess communication mechanisms and to prove the correctness of algorithms for implementing them.

Capsule Review by Andrei Broder

Concurrent systems are customarily described hierarchically, each level being intended to implement the level above it. On each level certain actions are considered atomic with respect to that level, although they decompose into a set of actions at a lower level. Furthermore there are cases when, for efficiency purposes, their components might be interleaved in time at a lower level with no loss of semantic correctess, despite the fact that the atomicity specified on the higher level is not respected. In this paper a very clean formalism is developed that allows a cohesive description of the different levels and axiomatic proofs of the implementation properties, without using the atomic action concept.

Capsule Review by Paul McJones

A common approach to dealing with concurrency is to introduce primitives allowing the programmer to think in terms of the more familiar sequential model. For example, database transactions and linguistic constructs for mutual exclusion such as the monitor give a process the illusion that there is no concurrency. In contrast, Part II of this paper presents the approach of designing and verifying algorithms that work in the face of manifest concurrency.

Starting from some seemingly minimal assumptions about the nature of communication between asynchronous processes, the author proposes a classification of twelve partially-ordered kinds of single-writer shared registers. He provides constructions for implementing many of these classes from "weaker" ones, culminating in a multi-value, single-reader, atomic register. The constructions are proved both informally and using the formalism of Part I.

Much of the paper is of a theoretical nature. However, its ideas are worth study by system builders. For example, its algorithms and verification techniques could be of use in designing a "conventional" synchronization mechanism (e.g. a semaphore) for a multiprocessor system. A more exciting possibility would be to extend its approach to the design of a higher level concurrent algorithm such as taking a snapshot of an online database.

Contents

Part I
Basic Formalism

This paper addresses what I believe to be fundamental questions in the theory of interprocess communication. Part I 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 to algorithms that implement certain interprocess communication mechanisms. Readers interested only in these mechanisms and not in the formalism can skip Part I and read only Sections 4 and 5 of Part II.

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 10:35AM 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, 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 Section 2, where the additional correctness condition needed to rule out this absurd implementation is stated.

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" [11, 15, 16] assume that each element in the sequence represents an indivisible action. Net models [2] and related formalisms [10, 12] 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 [7] and refined in [6], in which the primitive elements are

[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.

operation executions that are not assumed to be atomic. This formalism is described below; the reader is referred to [7] and [6] 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 [8]).

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 $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ 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 $\mathcal{S}$ a nonempty subset $\mu(A)$ of $\mathbf{E}$, such that for every pair of operation executions A and B of $\mathcal{S}$:*

$$\begin{aligned} A \longrightarrow B &\equiv \forall a \in \mu(A) : \forall b \in \mu(B) : a \longrightarrow b \\ A \dashrightarrow B &\equiv \exists a \in \mu(A) : \exists b \in \mu(B) : a \longrightarrow b \text{ or } a = b \end{aligned} \qquad (1)$$

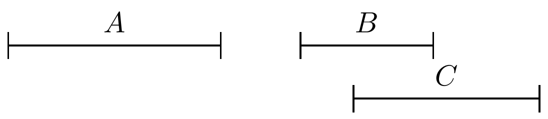

Figure 1: Three operation executions in a global-time model.

Note that the same symbol $\longrightarrow$ denotes the "precedes" relation both between operation executions in $\mathcal{S}$ 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* $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ *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 Figure 1, where operation executions 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 $\mathcal{S}$. Axiom A4 is illustrated (in a global-time model) by Figure 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 $B \not\dashrightarrow A$.

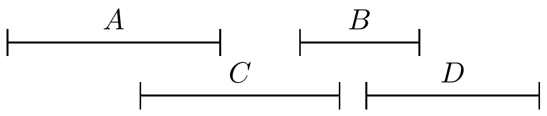

Figure 2: An illustration of Axiom A4.

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 $A \not\longrightarrow B$ is finite.

(These axioms differ from the ones in [6] 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 **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 system executions in terms of a global-time model, and to interpret the relations $\longrightarrow$ and $\dashrightarrow$ as indicated by the example in Figure 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 [5].) 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 $\mathcal{S}$ 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

[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.

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 $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ be a system execution and let $\mathcal{H}$ be a set whose elements, called *higher-level operation executions*, are sets of operation executions from $\mathcal{S}$. A model for $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ represents each operation execution in $\mathcal{S}$ by a set of events. This gives a representation of each higher-level operation execution H in $\mathcal{H}$ 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 $\overset{*}{\dashrightarrow}$, where $G \xrightarrow{*} H$ means that all events in (the representation of) G precede all events in H, and $G \overset{*}{\dashrightarrow} H$ means that some event in G precedes some event in H, for G and H in $\mathcal{H}$.

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

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

and define the precedence relations $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$ on $\mathcal{H}$ by

$$\begin{array}{rcl} G \xrightarrow{*} H & \equiv & \forall g \in \mu^*(G) : \forall h \in \mu^*(H) : g \longrightarrow h \\ G \overset{*}{\dashrightarrow} H & \equiv & \exists g \in \mu^*(G) : \exists h \in \mu^*(H) : g \longrightarrow h \text{ or } g = h \end{array}$$

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

$$\begin{array}{rcl} G \xrightarrow{*} H & \equiv & \forall A \in G : \forall B \in H : A \longrightarrow B \\ G \overset{*}{\dashrightarrow} H & \equiv & \exists A \in G : \exists B \in H : A \dashrightarrow B \text{ or } A = B \end{array} \qquad (2)$$

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

For the triple $\langle \mathcal{H}, \xrightarrow{*}, \overset{*}{\dashrightarrow} \rangle$ to be a system execution, the relations $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$ must satisfy axioms A1–A5. If each element of $\mathcal{H}$ 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 $\mathcal{H}$ consist of a finite number of elements of $\mathcal{S}$, and that each element of $\mathcal{S}$ belong to a finite number of elements of $\mathcal{H}$. 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* $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ *consists of a set* $\mathcal{H}$ *such that:*

H1. Each element of $\mathcal{H}$ is a finite, nonempty set of elements of $\mathcal{S}$.

H2. Each element of $\mathcal{S}$ belongs to a finite, nonzero number of elements of $\mathcal{H}$.

In most cases of interest, $\mathcal{H}$ is a partition of $\mathcal{S}$, so each element of $\mathcal{S}$ belongs to exactly one element of $\mathcal{H}$. 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 $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ is an implementation of a system execution $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$, then we expect $\mathcal{H}$ to be a higher-level view of $\mathcal{S}$—that is, each operation in $\mathcal{H}$ should consist of a set of operation executions of $\mathcal{S}$ satisfying H1 and H2. This describes the elements of $\mathcal{H}$, but not the precedence relations $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$. What should those relations be?

If we consider the operation executions in $\mathcal{S}$ to be the "real" ones, and the elements of $\mathcal{H}$ to be fictitious groupings of the real operation executions into abstract, higher-level ones, then the induced precedence relations $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$ represent the "real" temporal relations on $\mathcal{H}$. These induced relations make the higher-level view $\mathcal{H}$ a system execution, so they are an obvious choice for the relations $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$. 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

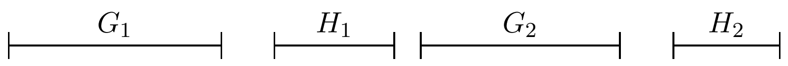

Figure 3: An example with $G \not\longrightarrow H$ and $H \not\longrightarrow G$.

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 $\langle \mathcal{H}, \overset{\mathcal{H}}{\longrightarrow}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$ of the database system, all the elements of $\mathcal{H}$ (the transactions) must be totally ordered by $\overset{\mathcal{H}}{\longrightarrow}$. This higher-level view of the database operations is implemented by lower-level operations that access individual database items. The higher-level system execution $\langle \mathcal{H}, \overset{\mathcal{H}}{\longrightarrow}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$ must be implemented by a lower-level one $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ in which each transaction H in $\mathcal{H}$ is implemented by a set of lower-level operation executions in $\mathcal{S}$.

Suppose $G = \{G_1, \ldots, G_m\}$ and $H = \{H_1, \ldots, H_n\}$ are elements of $\mathcal{H}$, where the G_i and H_i are operation executions in $\mathcal{S}$. For $G \overset{*}{\longrightarrow} H$ to hold, each G_i must precede ($\longrightarrow$) each H_j, and, conversely, $H \overset{*}{\longrightarrow} G$ only if each H_j precedes each G_i. In a situation like the one in Figure 3, neither $G \overset{*}{\longrightarrow} H$ nor $H \overset{*}{\longrightarrow} G$ holds. (For a system with a global-time model, this means that both $G \overset{*}{\dashrightarrow} H$ and $H \overset{*}{\dashrightarrow} G$ hold.) If we required that the relations $\overset{\mathcal{H}}{\longrightarrow}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ be the induced relations $\overset{*}{\longrightarrow}$ and $\overset{*}{\dashrightarrow}$, then the only way to implement a serializable system, in which $\overset{\mathcal{H}}{\longrightarrow}$ is a total ordering of the transactions, would be to prevent the type of interleaved execution shown in Figure 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.

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 $\overset{\mathcal{H}}{\longrightarrow}$ 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 $\overset{\mathcal{H}}{\longrightarrow}$.

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 $\xrightarrow{\mathcal{H}}$. 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 $\xrightarrow{\mathcal{H}}$ 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 $\xrightarrow{\mathcal{H}}$ 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 $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ to rule out precedence 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 $\overset{*}{\dashrightarrow}$ are the "real" precedence relations and $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ are the "pretend" ones, this leads to the following definition.

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

H3. For any $G, H \in \mathcal{H}$*: if* $G \xrightarrow{*} H$ *then* $G \xrightarrow{\mathcal{H}} H$*, 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 [5]. (Recall that a global-time model is determined by the mapping μ, since the set of events and their ordering is fixed.)

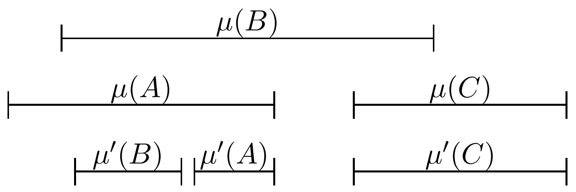

Figure 4: An illustration of Proposition 1.

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

This proposition is illustrated in Figure 4, where: (i) $\mathcal{S} = \{A, B, C\}$, (ii) $A \longrightarrow C$ is the only $\longrightarrow$ relation, and (iii) $B \xrightarrow{\prime} A \xrightarrow{\prime} C$. To apply Proposition 1 to Definition 5, substitute $\mathcal{S}$ for $\mathcal{H}$, substitute $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$ for $\longrightarrow$ and $\dashrightarrow$, and substitute $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ for $\xrightarrow{\prime}$ and $\overset{\prime}{\dashrightarrow}$. 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 $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$, whose operation executions consist of reads and writes of individual database items, must implement a system $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$, whose operations are database transactions, such that $\xrightarrow{\mathcal{H}}$ is a total ordering of $\mathcal{H}$. 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 $\xrightarrow{\mathcal{H}}$ that makes writes come long after they actually occurred.

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

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

- *$\mathcal{H} \cup \mathcal{T}, \xrightarrow{\mathcal{HT}}, \overset{\mathcal{HT}}{\dashrightarrow}$ is a system execution that is implemented by $\mathcal{S} \cup \mathcal{T}, \longrightarrow, \dashrightarrow$.*

- *The restrictions of $\xrightarrow{\mathcal{HT}}$ and $\overset{\mathcal{HT}}{\dashrightarrow}$ to $\mathcal{H}$ equal $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$, respectively.*

- *The restrictions of $\xrightarrow{\mathcal{HT}}$ and $\overset{\mathcal{HT}}{\dashrightarrow}$ to $\mathcal{T}$ are extensions of the relations $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$, respectively.*

To illustrate the significance of this proposition for Definition 5, let $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ be a system execution of reads and writes to database items that implements a higher-level system execution $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$ of database transactions. The operation executions of $\mathcal{S}$ presumably occur deep inside the computer and are not directly observable. Let $\mathcal{T}$ 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 $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ may imply new precedence relations on the operation executions in $\mathcal{T}$, these relations ($\xrightarrow{\mathcal{HT}}$ and $\overset{\mathcal{HT}}{\dashrightarrow}$) are consistent with the "real" precedence relations $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$ on $\mathcal{T}$. Thus, pretending that the database transactions occur in the order given by $\xrightarrow{\mathcal{H}}$ does not contradict any of the real, externally observable orderings among the operations in $\mathcal{T}$.

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 $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ of $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$ is said to be* trivial *if every element of $\mathcal{H}$ is a singleton set.*

In a trivial implementation, the sets $\mathcal{S}$ and $\mathcal{H}$ 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

[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

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 of **H** involve higher-level concepts such as transactions. The first thing we need is some way of interpreting a "concrete" system execution $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ 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 $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ of **S** a higher-level system execution $\iota(\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle)$ 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 [10] and [12]. 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

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 **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* $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ *in* **S**, $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ *implements* $\iota(\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle)$.

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 [9]. The specification must explicitly describe how certain interface operations are to be implemented; their implementation is not left to the implementor. 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 [9] for a discussion of this subject.

Part II
Algorithms

Part I describes a formalism for specifying and reasoning about concurrent systems. Here in Part II, communication between asynchronous processes in a concurrent system is studied. The next section explains why the problem of achieving asynchronous interprocess communication may be viewed as one of implementing shared registers, and the following section describes algorithms for doing this. These two sections are informal, and may be read without having read the formalism of Part I. The concepts introduced in Section 4 are formally defined in Section 6, and formal correctness proofs of the algorithms of Section 5 are given in Section 7. These latter two sections assume knowledge of the material in Part I.

4 The Nature of Asynchronous Communication

All communication ultimately involves a communication medium whose state is changed by the sender and observed by the receiver. A sending processor changes the voltage on a wire and a receiving processor observes the voltage change; a speaker changes the vibrational state of the air and a listener senses this change.

There are two kinds of communication acts: *transient* and *persistent*. In a transient communication act, the medium's state is changed only for the duration of the act, immediately afterwards reverting to its "normal" state. A message sent on an Ethernet modifies the transmission medium's state only while the message is in transit; the altered state of the air lasts only while the speaker is talking. In a persistent communication act, the state change remains after the sender has finished its communication. Setting a voltage level on a wire, writing on a blackboard, and raising a flag on a flagpole are all examples of persistent communication.

Transient communication is possible only if the receiver is observing the communication medium while the sender is modifying it. This implies an *a priori* synchronization—the receiver must be waiting for the communication to take place. Communication between truly asynchronous processes must be persistent, the sender changing the state of the medium and the receiver able to sense that change at a later time.

At a low level, message passing is often considered to be a form of transient communication between asynchronous processes. However, a closer examination of asynchronous message passing reveals that it involves a persistent communication. Messages are placed in a buffer that is periodically tested by the receiver. Viewed at a low level, message passing is typically accomplished by putting a message in a buffer and setting an interrupt bit that is tested on every machine instruction. The receiving process actually consists of two asynchronous subprocesses: a *main* process that is usually thought of as the receiver, and an *input* process that continuously monitors the communication medium and transfers messages from the medium to the buffer. The input process is synchronized with the sender (it is a "slave" process) and communicates asynchronously with the main process, using the buffer as a medium for persistent communication.

The subject of this paper is asynchronous interprocess communication, so only persistent communication is considered. Moreover, attention is restricted to unidirectional communication, in which only a single process can modify the state of the medium. (With this restriction, two-way communication requires at least two separate communication media, one modified by each process.) However, multiple receivers will be considered. Also, only discrete systems, in which the medium has a finite number of distinguishable states, are considered. A receiver is assumed always to obtain one of these discrete values. The sender can therefore set the medium to one of a fixed number of persistent states, and the receiver(s) can observe the medium's state.

This form of persistent communication is more commonly known as a shared register, where the sender and receiver are called the *writer* and *reader*, respectively, and the state of the communication medium is known as the *value* of the register. These terms are used in the rest of this paper, which therefore considers finite-valued registers with a single writer and one or more readers.

In assuming a single writer, the possibility of concurrent writes (to the same register) is ruled out. Since a reader only senses the value of the register, there is no reason why a read operation must interfere with another read or write operation. (While reads do interfere with other operations in some forms of memory, such as magnetic core, this interference is an idiosyncrasy of the particular technology rather than an inherent property of reading.) A read is therefore assumed not to affect any other read or any write. However, it is not clear what effect a concurrent write should have on a read.

In concurrent programming, one traditionally assumes that a writer has exclusive access to shared data, making concurrent reading and writing impossible. This assumption is enforced either by requiring the programming language to provide the necessary exclusive access, or by implementing the exclusion with a "readers-writers" protocol [3]. Such an approach requires that a reader wait while a writer is accessing the register, and vice versa. Moreover, any method for achieving such exclusive access, whether implemented by the programmer or the compiler, requires a lower-level shared register. At some level, the problem of concurrent access to a shared register must be faced. It is this problem that is addressed by this paper; any approach that requires one process to wait for another is eschewed.

Asynchronous concurrent access to shared registers is usually considered only at the hardware level, so it is at this level that the methods developed here could have some direct application. However, concurrent access to shared data also occurs at higher levels of abstraction. One cannot allow any single process exclusive access to the entire Social Security system's database. While algorithms for implementing a single register cannot be applied to such a database, I hope that insight obtained from studying these algorithms will eventually lead to new methods for higher-level data sharing. Nevertheless, when reading this paper, it is best to think of a register as a low-level component, probably implemented in hardware.

Hardware implementations of asynchronous communication often make assumptions about the relative speeds of the communicating processes. Such assumptions can lead to simplifications. For example, the problem of constructing an atomic register, discussed below, is shown to be easily solved by assuming that two successive reads of a register cannot be concurrent with a single write. If one knows how long a write can take, a delay can be added between successive reads to ensure that this assumption holds. No such assumptions are made here about process speeds. The results therefore apply even to communication between processes of vastly differing speeds.

Writes cannot overlap (be concurrent with) one another because there is only one writer, and overlapping reads are assumed not to affect one another, so the only case left to consider is a read overlapping one or more writes. Three possible assumptions about what can happen in this case are considered.

The weakest possibility is a *safe* register, in which it is assumed only that a read not concurrent with any write obtains the correct value—that is, the most recently written one. No assumption is made about the value obtained by a read that overlaps a write, except that it must obtain one of the possible values of the register. Thus, if a safe register may assume

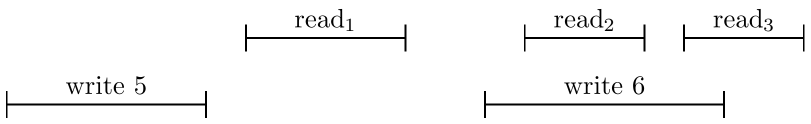

Figure 5: Two writes and three reads.

the values 1, 2, and 3, then any read must obtain one of these three values. A read that overlaps a write operation that changes the value from 1 to 2 could obtain any of these values, including 3.

The next stronger possibility is a *regular* register, which is safe (a read not concurrent with a write gets the correct value) and in which a read that overlaps a write obtains either the old or new value. For example, a read that overlaps a write that changes the value from 1 to 3 may obtain either 1 or 3, but not 2. More generally, a read that overlaps any series of writes obtains either the value before the first of the writes or one of the values being written.

The final possibility is an *atomic* register, which is safe and in which reads and writes behave as if they occur in some definite order. In other words, for any execution of the system, there is some way of totally ordering the reads and writes so that the values returned by the reads are the same as if the operations had been performed in that order, with no overlapping. (The precise formal condition was developed in Section 2 of Part I.)

The difference between the three kinds of registers is illustrated by Figure 5, which shows five operations to a register that may assume the three values 5, 6, and 27. The duration of each operation is indicated by a line segment, where time runs from left to right. A write of the value 5 precedes all other operations, including a subsequent write of 6. There are three successive reads, denoted read$_1$, read$_2$, and read$_3$.

For a safe register, read$_1$ obtains the value 5, since a read that does not overlap a write must obtain the most recently written value. However, the other two reads, which overlap the second write, may obtain 5, 6, or 27.

With a regular register, read$_1$ must again obtain the value 5, since a regular register is also safe. Each of the other two reads may obtain either a 5 or a 6, but not a 27. In particular, read$_2$ could obtain a 6 and read$_3$ a 5.

With an atomic register, read$_1$ must also obtain the value 5 and the other two reads may obtain the following pairs of values:

read_2	read_3
5	5
5	6
6	6

For example, the pair of values 5,6 represents a situation in which the operations act as if the first read preceded the write of 6 and the second read followed it. However, unlike a regular register, an atomic register does not admit the possibility of read_2 obtaining the value 6 and read_3 obtaining 5. In general, if two successive reads overlap the same write, then a regular register allows the first read to obtain the new value and the second read the old value, while this is forbidden with an atomic register. In fact, Proposition 5 of Section 6 essentially states that a regular register is atomic if two successive reads that overlap the same write cannot obtain the new then the old value. Thus, a regular register is automatically an atomic one if two successive reads cannot overlap the same write.

These are the only three general classes of register that I have been able to think of. Each class merits study. Safeness[5] seems to be the weakest requirement that allows useful communication; I do not know how to achieve any form of interprocess synchronization with a weaker assumption. Regularity asserts that a read returns a "reasonable" value, and seems to be a natural requirement. Atomicity is the most common assumption made about shared registers, and is provided by current multiport computer memories.[6] At a lower level, such as interprocess communication within a single chip, only safe registers are provided; other classes of register must be implemented using safe ones.

Any method of implementing a single-writer register can be classified by three "coordinates" with the following values:

- *safe*, *regular*, or *atomic*, according to the strongest assumption that the register satisfies.

- *boolean* or *multivalued*, according to whether the method produces only boolean registers or registers with any desired number of values.

[5]The term "safeness" is used because "safety" already has a technical meaning for concurrent programs.

[6]However, the standard implementation of a multiport memory does not meet my requirements for an asynchronous register because, if two processes concurrently access a memory cell, one must wait for the other.

- *single-reader* or *multireader*, according to whether the method yields registers with only one reader or with any desired number of readers.

This produces twelve classes of implementations, partially ordered by "strength"—for example, a method that produces atomic, multivalued, multireader registers is stronger than one producing regular, multivalued, single-reader registers. This paper addresses the problem of implementing a register of one class using one or more registers of a weaker class.

The weakest class of register, and therefore the easiest to implement, is a safe, boolean, single-reader one. This seems to be the most natural kind of register to implement with current hardware technology, requiring only that the writer set a voltage level either high or low and that the reader test this level without disturbing it.[7] A series of constructions of stronger registers from weaker ones is presented that allows almost every class of register to be constructed starting from this weakest class. The one exception is that constructing an atomic, multireader register from any weaker one is still an open problem. Most of the constructions are simple; the difficult ones are Construction 4 that implements an m-reader, multivalued, regular register using m-reader, boolean, regular registers, and Construction 5 that implements a single-reader, multivalued, atomic register using single-reader, multivalued, regular registers.

5 The Constructions

In this section, the algorithms for constructing different classes of registers are described and informally justified. Rigorous correctness proofs are postponed until Section 7.

The algorithms are described by indicating how a write and a read are performed. For most of them, the initial state is not indicated—it is the one that would result from writing the initial value starting from any arbitrary state.

The first construction implements a multireader safe or regular register from single-reader ones. It uses the obvious method of having the writer maintain a separate copy of the register for each reader. The **for all** statement denotes that its body is executed once for each of the indicated values of i; these separate executions can be done in any order or concurrently.

[7]This is only safe and not regular if, for example, setting a level high when it is already high can cause a perturbation of the level.

Construction 1 *Let $v_1, \ldots, v_m$ be single-reader, n-valued registers, where each v_i can be written by the same writer and read by process i, and construct a single n-valued register v in which the operation $v := \mu$ is performed as follows:*

```
**for all** $i$ **in** $\{1, \ldots, m\}$ **do** $v_i := \mu$ **od**
```

and process i reads v by reading the value of v_i. If the v_i are safe or regular registers, then v is a safe or regular register, respectively.

The proof of correctness for this construction runs as follows. Any read by process i that does not overlap a write of v does not overlap a write of v_i. If v_i is safe, then this read gets the correct value, which shows that v is safe. If a read of v_i by process i overlaps a write of v_i, then it overlaps the write of the same value to v. This implies that if v_i is regular, then v is also regular.

Construction 1 does not make v an atomic register even if the v_i are atomic. If reads by two different processes i and j both overlap the same write, it is possible for i to get the new value and j the old value even though the read by i precedes the read by j—a possibility not allowed by an atomic register.

The next construction is also trivial; it implements an n-bit safe register from n single-bit ones.

Construction 2 *Let $v_1, \ldots, v_n$ be boolean m-reader registers, each written by the same writer and read by the same set of readers. Let v be the 2^n-valued, m-reader register in which the number with binary representation $\mu_1 \ldots \mu_n$ is written by*

```
**for all** $i$ **in** $\{1, \ldots, m\}$ **do** $v_i := \mu_i$ **od**
```

and in which the value is read by reading all the v_i. If each v_i is safe, then v is safe.

This construction yields a safe register because, by definition, a read does not overlap a write of v only if it does not overlap a write of any of the v_i, in which case it obtains the correct values. The register v is not regular even if the v_i are. A read can return any value if it overlaps a write that changes the register's value from $0 \ldots 0$ to $1 \ldots 1$.

The next construction shows that it is trivial to implement a boolean regular register from a safe boolean register. In a safe register, a read that overlaps a write may get any value, while in a regular register it must get either the old or new value. However, a read of a safe boolean register must obtain either *true* or *false* on any read, so it must return either the old or new value if it overlaps a write that changes the value. A boolean safe register can fail to be regular only if a read that overlaps a write that does not change the value returns the other value—for example, writing the value *true* when the current value equals *true* could cause an overlapping read to obtain the value *false*. To prevent this possibility, one simply does not perform a write that does not change the value.

Construction 3 *Let v be an m-reader boolean register, and let x be a variable internal to the writer (not a shared register) initially equal to the initial value of v. Define v^* to be the m-reader boolean register in which the write operation $v^* := \mu$ is performed as follows:*

```
if $x \neq \mu$ then $v := \mu$;
                     $x := \mu$  fi
```

and a read of v^* is performed by reading v. If v is safe then v^* is regular.

There are two known algorithms for implementing a multivalued regular register from boolean ones. The simpler one is given as Construction 4; the second one is described later. Construction 4 employs a unary encoding, in which the value μ is denoted by zeros in bits 0 through $\mu - 1$ and a one in bit μ. A reader reads the bits from left to right (0 to n) until it finds a one. To write the value μ, the writer first sets v_μ to one and then sets bits $\mu - 1$ through 1 to zero, writing from right to left. (While this algorithm has never before been published, the idea of implementing shared data by reading and writing its components in different directions was also used in [4].[8])

Construction 4 *Let $v_1, \ldots, v_n$ be boolean, m-reader registers, and let v be the n-valued, m-reader register in which the operation $v := \mu$ is performed by*

```
$v_\mu := 1$;
for $i := \mu - 1$ step $-1$ until 1  do $v_i := 0$ od
```

[8]Although the algorithms in [4] require only that the registers be regular, the assumption of atomicity was added because the editor felt that nonatomicity at the level of individual bits was too radical a concept to appear in *Communications of the ACM*.

and a read is performed by:

```
$\mu := 1$;
while $v_\mu = 0$ do $\mu := \mu + 1$ od;
return $\mu$
```

If each v_i is regular, then v is regular.

The correctness of this algorithm is not at all obvious. Indeed, it is not even obvious that the **while** loop in the read operation does not "fall off the end" and try to read the nonexistent register v_{n+1}. This can't happen because, whenever the writer writes a zero, there is a one to the right of it. (Since an initial value is assumed to have been written, some v_i initially equals one.) As an exercise, the reader of this paper can convince himself that, whenever a reading process sees a one, it was written by either a concurrent write or by the most recent preceding one, so v is regular. The formal proof is given in Section 7.

The value of v_n is only set to one, never to zero. It can therefore be eliminated; the writer simply never writes it and the reader assumes its value is one instead of reading it.

Even if all the v_i are atomic, Construction 4 does not produce an atomic register. To see this, suppose that the register initially has the value 3, so $v_1 = v_2 = 0$ and $v_3 = 1$, the writer first writes the value 1 then the value 2, and there are two successive read operations. This can produce the following sequence of actions:

- the first read finds $v_1 = 0$
- the first write sets $v_1 := 1$
- the second write sets $v_2 := 1$
- the first read finds $v_2 = 1$ and returns the value 2
- the second read finds $v_1 = 1$ and returns the value 1.

In this scenario, the first read obtains a newer value (the one written by the second write) than the second read (which obtains the one written by the first write), even though it precedes the second read. This shows that the register is not atomic.

Construction 4 uses $n - 1$ boolean regular registers to make an n-valued one, so it is practical only for small values of n. One would like an algorithm that requires $O(\log n)$ boolean registers to construct an n-valued register. The second method for constructing a regular multivalued register uses an algorithm of Peterson [14] that implements an m-reader, n-valued, atomic register with $m+2$ safe, m-reader, n-valued registers; $2m$ atomic, boolean, one-reader registers; and two atomic, boolean m-reader registers. However, there is no known algorithm for constructing the atomic, m-reader registers required by Peterson's algorithm from simpler ones. Nevertheless, we can apply his algorithm to construct an n-valued, single-reader, atomic register using three safe, single-reader, n-valued registers and four single-reader, atomic, boolean registers. The safe registers can be implemented with Construction 2, and the atomic boolean registers can be implemented with Construction 5 below. Since an atomic register is regular, Construction 1 can then be used to make an m-reader, n-valued, regular register from $O(3m \log n)$ single-reader, boolean, regular registers.

Before giving the algorithm for constructing a two-reader atomic register, a result is proved that indicates why no trivial algorithm will work. It asserts that there can be no algorithm in which the writer only writes and the reader only reads; any algorithm must involve two-way communication between the reader and the writer.

Theorem: *There exists no algorithm to implement an atomic register using a finite number of regular registers that can be written only by the writer (of the atomic register).*

Proof: We assume such an algorithm and derive a contradiction. Any algorithm that uses multiple registers can be replaced by one in which these registers are combined into a single large register. A read in the original algorithm is replaced by one that reads all the combined register and ignores the other components; a write in the original algorithm is replaced by one that changes only the desired component of the combined register. (This is possible because there is only a single writer.) Therefore, without loss of generality, we can assume that there is only a single regular register v written by the writer and read by the reader.

Let v^* denote the atomic register that is being implemented. Since the algorithm must work if the writer never stops writing, we may suppose that the writer performs an infinite number of writes that change the value of v^*. There must be some pair of values assumed by v^*, call them 0 and 1, such that there are an infinite number of writes that change v^*'s value from 0 to 1. Since v can assume only a finite number of values (the hypothesis states that the original algorithm has only a finite number of registers, and all registers are taken to have only a finite number of possible values), there must exist values $v_0, \ldots, v_n$ of v such that: (i) v_0 is the final value of v after each one of an infinite number of writes of 0 to v^*, (ii) v_n is the final value of v after each one of an infinite number of writes of 1 to v^*, and (iii) for each $i < n$, the value of v is changed from v_i to v_{i+1} during infinitely many writes that change the value of v^* from 0 to 1.[9]

A read of v^* may involve several reads of v. However, in our quest for a contradiction, we may restrict our attention to scenarios in which each of those reads of v obtains the same value, so we may assume that each read of v^* reads v only once. Since v assumes each value v_i infinitely often, it must be possible for a sequence of $n+1$ consecutive reads of v to obtain the values $v_n, v_{n-1}, \ldots, v_0$.

The read that finds v equal to v_i and the subsequent read that finds v equal to v_{i-1} could both have overlapped the same write of v, which could have been a write that occurred in the process of changing v^*'s value from 0 to 1. Therefore, if the read of v^* that finds v equal to v_i returns the value 1, then the subsequent read that finds v equal to v_{i-1} must also return the value 1, since both reads could be overlapping the same write and, in that case, two successive reads of an atomic register cannot return first the new value, then the old one.

The first read, which finds v equal to v_n, must return the value 1, since it could have occurred after the completion of a write of 1. By induction, this implies that the last read, which found v equal to v_0, must return the value 1. However, this read could have occurred after a write of 0 and before any subsequent write, so returning the value 1 would violate the assumption that the register v^* is safe. (An atomic register is *a fortiori* safe.) This is the required contradiction. ∎

.

This theorem could be expressed and proved using the formalism of Part I and the definitions of the next section, but doing so would lead to no new insight. The formalization of this theorem is therefore left as an exercise for the reader who wishes to gain practice in using the formalism.

The theorem is false if no bound is placed on the number of values a register can hold. Given a regular register v that can assume an unbounded

[9]If we assume that the writer has only a finite number of internal states, then we can conclude that the precise sequence of values $v_0, \ldots, v_n$ is written infinitely many times when changing the value of v^* from 0 to 1. However, with an infinite number of internal states, it is possible for the writer never to perform the same sequence of writes to v twice.

number of values, an atomic register v^* is implemented as follows. The writer sets v equal to a pair consisting of the value of v^* and a sequential version number. The reader reads v and compares the version number with the previous one it read. If the new version number is higher, then it uses the value it just read; if the new version number is lower, then it forgets the value and version number it just read and uses the previously read value. The correctness of this algorithm follows easily from Proposition 5 of Section 6. By assuming that registers hold only a bounded set of values, such algorithms are disallowed.

Finally, we come to the algorithm for constructing a single-reader, multi-valued, atomic register from regular ones. Let v^* denote the atomic register being implemented, and let the writer set this register by writing into a shared regular register v. Suppose that some value μ of v^* is represented by letting v equal v_0, and that to change the value of v^* to another value ν, the writer successively sets v to the values v_1, v_2, $\ldots$, v_n, where $v = v_n$ represents $v^* = \nu$. The proof of the above theorem rested upon showing that the reader is in a quandary if n successive reads return the values v_n, v_{n-1}, $\ldots$, v_0. If the i^{th} read returns ν then the $i+1^{\text{st}}$ read must also return ν because both reads could have overlapped the same write of v, in which case returning μ would result in the later read returning the earlier value. The first read must return the value ν, so the last read, which ought to return μ, the value of * denoted by $v = v_0$, is forced to return ν.

The way out of this problem is to encode, as part of v's value, a boolean quantity called a *color*. Each value of v^* is represented by two different values of v—one of each color. Every time the reader reads v, it sets a boolean register c to the color of the value it just read. When the writer wants to write a new value of v^*, it first reads c and then makes the series of values v_1, $\ldots$, v_n it writes to v have the opposite color to c. (Thus, the reader tries to keep c equal to the color of v, and the writer tries to keep the color of v different from c.) It can be shown that if $n \geq 4$, so at least three intermediate values are written when changing the value of v^*, then successive reads cannot obtain the sequence v_n, $\ldots$, v_0. This enables one to devise an algorithm in which the writer changes the value of the register from μ to ν by first writing a series of intermediate values $(\mu, \nu, 1, \kappa)$, $(\mu, \nu, 2, \kappa)$, $(\mu, \nu, 3, \kappa)$, and then writing (ν, κ), where κ is the color. However, one can do better, and an algorithm is developed below that uses only two intermediate values.

When $n = 3$, so the writer writes the sequence v_1, v_2, v_3, with v_3 representing the new value ν, it is possible for three successive reads R_3, R_2,

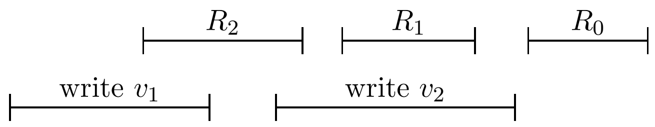

Figure 6: Reads R_2 and R_1 overlapping the write of v_2.

R_1 to obtain the values v_3, v_2, and v_1, respectively. However, it will be shown that this can happen only if the two reads R_2 and R_1 both overlap a single write of v_2. As indicated by Figure 6, this implies that a fourth read R_0 cannot overlap the write of the value v_1 that was obtained by R_1. Therefore, if the fourth read R_0 obtains v_0, the reader can return the value μ represented by v_0 with no fear of the pair R_1, R_0 returning a forbidden "new-old" sequence.

The following construction implements an atomic register v^* using two regular registers v (written by the writer) and c (written by the reader). For clarity, it is presented in a form in which v can assume more values than are necessary; the number of different values that v really needs is discussed afterwards. To change the value of v^* from μ to ν, the writer first sets nc to be different from c, then writes the following sequence of values to v: $(\mu, \nu, 1, nc)$, $(\mu, \nu, 2, nc)$, $(\mu, \nu, 3, nc)$. Thus, $v = (\mu, \nu, 3, \kappa)$ denotes $v^* = \nu$ for any values of μ and κ.

The reader reads v and sets c equal to its color, but what value of v^* does it return? Suppose the reader obtains the value (μ, ν, i, κ) when reading v. If $i = 3$, then to guarantee safeness, the reader must return ν. If $i < 3$, then regularity requires only that the read return either μ or ν. The basic idea is for the reader to return μ except when this might allow the possibility that two successive reads overlapping the same write return first the new then the old value. For example, this is the case if the preceding read had obtained the value $(\mu, \nu, i + 1, \kappa)$ and returned the value ν. To simplify the algorithm, the reader bases its decision of which value to return only upon the values of i and κ obtained by this and the preceding read, not upon the values of μ and ν obtained by the preceding read.

The following notation is used in describing the algorithm: for $\xi = (\mu, \nu, i, \kappa)$, let $old(\xi) = \mu$, $new(\xi) = \nu$, $num(\xi) = i$, and $col(\xi) = \kappa$. In the algorithm, the variable v is written by the writer and read by both the reader and the writer. A two-reader register is not needed, since the writer can maintain a local variable containing the value that it last wrote into v. (This is just Construction 1 with $m = 2$ and the writer being the second reader.) Such a local variable would complicate the description, so it is omitted. The variables nc, μ, rv, rv', and $nuret$ are local (not shared registers); $nuret$ is true if the reader returned the "ν value" on the preceding read. The proof of correctness of this construction is given in Section 7.

Construction 5 *Let w and r be processes, let $\mathcal{V}^*$ be a finite set, let v be a regular register with values in $\mathcal{V}^* \times \mathcal{V}^* \times \{1,2,3\} \times \{true, false\}$ that can be written by w and read by r, with $num(v)$ initially equal to 3, and let c be a regular boolean register that can be written by r and read by w. Define the register v^* with values in $\mathcal{V}^*$, written by w and read by r, as follows. The write $v^* := \nu$ is performed by*

```
$nc := \neg c$;
$\mu$  $:= old(v)$;
**for** $i := 1$ **until** 3 **do** $v := (\mu, \nu, i, nc)$
```

and the read operation is performed by the following algorithm, where nuret is initially false:

```
$rv' := rv$;
$rv$  $:= v$;
$c$   $:= col(rv)$;
**if** $num(rv)=3$
   **then** $nuret := true$;
          return $new(rv)$
   **else** **if** $nuret \wedge col(rv) = col(rv') \wedge num(rv) \geq num(rv') - 1$
             **then** return $new(rv)$
             **else**  $nuret := false$;
                    return $old(rv)$
**fi**         **fi**
```

Then v^* is an atomic register.

If a read R_1^* of v^* obtains the value $(\mu, \nu, 1, \kappa)$ for v and returns ν as the value of v^*, then there must have been two previous reads R_3^* and R_2^* that obtained the values $(\ldots, 3, \kappa)$ and $(\ldots, 2, \kappa)$, respectively, for v such that any reads coming between R_3^* and R_1^* obtained a value $(\ldots, \kappa)$. It will be shown in the correctness proof of the construction that this can happen only if R_2^* obtained the value $(\mu, \nu, 2, \kappa)$. This means that the read R_1^* can simply return the same value returned by R_2^*. Hence, if the reader remembers the last value returned by a read that found $num(rv) = 2$, then the ν component is redundant in values of v of the form $(\mu, \nu, 1, \kappa)$.

When $num(rv) = 3$, the reader always returns $new(rv)$. Hence, the μ component is redundant in values of v of the form $(\mu, \nu, 3, \kappa)$. Since the writer can simply do nothing if the value it is writing is the same as the current value, there is no need for v to assume values in which $\mu = \nu$.

From these observations, it follows that v need assume only values of the following forms, with $\mu \neq \nu$: $(\mu, 1, \kappa)$, $(\mu, \nu, 2, \kappa)$, and $(\nu, 3, \kappa)$. If there are n possible values for μ and ν, then there are $2n(n+2)$ such values. Therefore, Construction 5 can be modified to implement an n-valued atomic register v^* with a $2n(n+2)$-valued regular register v written by the writer and read by the reader and a boolean regular register c written by the reader and read by the writer.

6 Register Axioms

The formalism described in Part I applies to any system execution. For system executions containing reads and writes to registers, the general axioms A1–A5 of Part I must be augmented by axioms for these operation executions. They include axioms that provide the formal statements of the properties of safe, regular, and atomic registers.

Axioms A1–A5 do not require that there be any precedence relations among operation executions. However, some precedence relations must be assumed among operations to the same register. Implicit in our assumption that a register has only a single writer is the assumption that all the writes to a register are totally ordered. We let $V^{[1]}$, $V^{[2]}$, ... denote the sequence of write operations to the register v, where $V^{[1]} \longrightarrow V^{[2]} \longrightarrow \cdots$ and let $v^{[i]}$ denote the value written by $V^{[i]}$. (There may be a finite or infinite number of write operations $V^{[i]}$.)

A register v is assumed to have some initial value $v^{[0]}$. It is convenient to assume that this value is written by a write $V^{[0]}$ that precedes ($\longrightarrow$) all other reads and writes of v. Eliminating this assumption changes none of the results, but it complicates the reasoning because a read that precedes all writes has to be treated as a separate case. These assumptions are expressed formally by the following axiom.

B0. The set of write operation executions to a register v consists of the (finite or infinite) set $\{V^{[0]}, V^{[1]}, \ldots\}$ where $V^{[0]} \longrightarrow V^{[1]} \longrightarrow \cdots$ and,

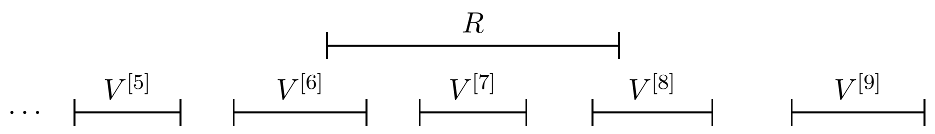

Figure 7: A read that sees $v^{[5,8]}$.

for any read R of v, $V^{[0]} \longrightarrow R$. The value written by $V^{[i]}$ is denoted $v^{[i]}$.

Communication implies causal connection; for processes to communicate through operations to a register, there must be some causality ($\dashrightarrow$) relations between reads and writes of the register. The following axiom is therefore assumed; the reader is referred to [6] (where it is labeled C3) for its justification.

B1. For any read R and write W to the same register, $R \dashrightarrow W$ or $W \dashrightarrow R$ (or both).

Note that B1 holds for any system execution that has a global-time model because, for any operation executions A and B in such a system execution, either $A \longrightarrow B$ or $B \dashrightarrow A$.

Each register has a finite set of possible values—for example, a boolean-valued register has the possible values *true* and *false*. A read is assumed to obtain one of these values, whether or not it overlaps a write.

B2. A read of a register obtains one of the (finite collection of) values that may be written in the register.

Thus, a read of a boolean register cannot obtain a nonsense value like "*trlse*". Axiom B2 does not assert that the value obtained by a read was ever actually written in the register, so it does not imply safeness.

Let R be a read of register v, and let

$$\begin{aligned} I_R &\overset{\text{def}}{=} \{V^{[k]} : R \not\dashrightarrow V^{[k]}\} \\ J_R &\overset{\text{def}}{=} \{V^{[k]} : V^{[k]} \dashrightarrow R\} \end{aligned}$$

In the example of Figure 7, $I_R = \{V^{[0]}, \ldots, V^{[5]}\}$ and $J_R = \{V^{[0]}, \ldots, V^{[8]}\}$. As this example shows, in system executions with a global-time model, I_R is the set of writes that precede ($\longrightarrow$) R and the writes in J_R are the ones that could causally affect R. The difference $J_R - I_R$ of these two sets is the set of writes that are concurrent with (overlap) R. If we think of the register as containing "traces" of both the old and new values during a write, then a read R can see traces of the values written by writes in $J_R - I_R$ and by the last write in I_R. In Figure 7, R can see traces of the values $v^{[5]}$ through $v^{[8]}$. (The value $v^{[5]}$ is present during the write $V^{[6]}$, which is overlapped by R.) All traces of earlier writes vanish with the completion of the last write in I_R, and R sees no value written after the last write in J_R. This suggests the following formal definition, where "sees $v^{[i,j]}$" is an abbreviation for "sees traces of $v^{[i]}$ through $v^{[j]}$".

Definition 9 *A read R of register v is said to* see $v^{[i,j]}$ *where:*

$$\begin{aligned} i &\overset{\text{def}}{=} \max\{k : R \not\dashrightarrow V^{[k]}\} \\ j &\overset{\text{def}}{=} \max\{k : V^{[k]} \dashrightarrow R\} \end{aligned}$$

The informal discussion that led to this definition was based upon a global-time model. When the existence of a global-time model is not assumed, I_R not only contains all the writes that precede R, but it may contain later writes as well. The set J_R consists of all writes that could causally affect R.

For Definition 9 to make sense, it is necessary that the sets whose maxima are taken—or, equivalently, the sets I_R and J_R—be finite and nonempty. They are nonempty because, by A2 and the assumption that $V^{[0]}$ precedes all reads, both I_R and J_R contain $V^{[0]}$; and Axioms A5 and A2 imply that they are finite. Furthermore, B1 implies that $I_R \subseteq J_R$, so $i \leq j$.

The formal definitions of safe, regular, and live registers can now be given. A safe register has been informally defined to be one that obtains the correct value if it is not concurrent with any write. A read that is not concurrent with a write is one that sees traces of only a single write, which leads to the following definition:

B3. (*safe*) A read that sees $v^{[i,i]}$ obtains the value $v^{[i]}$.

A regular register is one for which a read obtains a value that it "could have" seen—that is, a value it has seen a trace of.

B4. (*regular*) A read that sees $v^{[i,j]}$ obtains a value $v^{[k]}$ for some k with $i \leq k \leq j$.

An atomic register satisfies the additional requirement that a read is never concurrent with any write.

B5. (*atomic*) If a read sees $v^{[i,j]}$ then $i = j$.

A safe register satisfies B0–B3, a regular register satisfies B0–B4 (note that B4 implies B3), and an atomic register satisfies B0–B5.

Observe that in B3–B5, the conditions placed upon the value obtained by a read R of register v depend only upon precedence relations between R and writes of v. No other operation executions affect R. In particular, a read is not influenced by other reads.

The following two propositions state some useful properties that are simple consequences of Definition 9. In Proposition 3, the notation is introduced that $v^{[i,j]}$ denotes a read that sees the value $v^{[i,j]}$, so part (a) is an abbreviation for: "If R is a read that sees $v^{[i,j]}$ and $R \longrightarrow V^{[k]}$, then ...." (Recall that $V^{[k]}$ is the k^{th} write of v.)

Proposition 3 (a) *If $v^{[i,j]} \longrightarrow V^{[k]}$ then $j < k$.*

(b) *If $V^{[k]} \longrightarrow v^{[i,j]}$ then $k \leq i$.*

(c) *If $v^{[i,j]} \longrightarrow v^{[i',j']}$ then $j \leq i' + 1$.*

Proof: Parts (a) and (b) are immediate consequences of Definition 9. To prove part (c), observe first that Definition 9 also implies that $V^{[j]} \dashrightarrow v^{[i,j]}$. Part (c) is immediate if $j = 0$. If $j > 0$, then $V^{[j-1]} \longrightarrow V^{[j]}$. Combining these two relations with the hypothesis gives

$$V^{[j-1]} \longrightarrow V^{[j]} \dashrightarrow v^{[i,j]} \longrightarrow v^{[i',j']}$$

Axiom A4 implies that $V^{[j-1]} \longrightarrow v^{[i',j']}$, which, by A2, implies $v^{[i',j']} \not\dashrightarrow V^{[j-1]}$. Definition 9 then implies that $j - 1 \leq i'$. ∎

Proposition 4 *If R is a read that sees $v^{[i,j]}$, then*

(a) *$k \leq j$ if and only if $V^{[k]} \dashrightarrow R$.*

(b) *$i \leq k$ if and only if $R \dashrightarrow V^{[k+1]}$.*

Proof: To prove part (a), observe that it follows immediately from Definition 9 that $V^{[k]} \dashrightarrow R$ implies $k \leq j$. To prove the converse, assume $k \leq j$. Since $V^{[j]} \dashrightarrow R$, the desired conclusion, $V^{[k]} \dashrightarrow R$, is immediate if $k = j$. If $k < j$, then $V^{[k]} \longrightarrow V^{[j]}$, and the result follows from A3.

For part (b), Definition 9 implies that if $i < k'$ then $R \dashrightarrow V^{[k']}$. Letting $k' = k+1$, this shows that if $i \leq k$ then $R \dashrightarrow V^{[k+1]}$. Conversely, suppose $R \dashrightarrow V^{[k+1]}$. By Definition 9, this implies $k+1 \neq i$. If $k+1 < i$, then $V^{[k+1]} \longrightarrow V^{[i]}$, so A3 would imply $R \dashrightarrow V^{[i]}$, contrary to Definition 9. Hence, we must have $i < k+1$, so $i \leq k$, completing the proof of part (b). ∎

Atomicity is usually taken to mean that all reads and writes are totally ordered in time. With B5, atomicity is defined by the requirement that each individual read is totally ordered with respect to the writes, but it leaves the possibility that two reads may overlap. It can be shown that, given a system execution for an atomic register, the partial ordering $\longrightarrow$ can be completed to a total ordering of reads and writes without violating conditions B1–B5. Thus, a system containing an atomic register trivially implements one in which all reads and writes are sequentially ordered. (Recall the definition of a trivial implementation in Part I.)

The following proposition is used in the formal correctness proof of Construction 5.

Proposition 5 *Let $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ be a system execution containing reads and writes to a regular register v. If there exists an integer-valued function ϕ on the set of reads such that:*

1. *If R sees $v^{[i,j]}$, then $i \leq \phi(R) \leq j$.*

2. *A read R returns the value $v^{[\phi(R)]}$.*

3. *If $R \longrightarrow R'$ then $\phi(R) \leq \phi(R')$.*

then $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ trivially implements a system execution in which v is an atomic register.

Proof: Proposition 2 of Part I, with the set of reads and writes of v substituted for $\mathcal{S}$ and with the set of all other operations in $\mathcal{S}$ substituted for $\mathcal{T}$, shows that it suffices to prove the proposition under the assumption that $\mathcal{S}$ consists entirely of the reads and writes of v.

Let $\xrightarrow{1}$ be the relation on $\mathcal{S}$ that is the same as $\longrightarrow$ except between reads and writes of v, and, for any read R and write $V^{[k]}$ of v: $V^{[k]} \xrightarrow{1} R$ if $k \leq \phi(R)$, and $R \xrightarrow{1} V^{[k]}$ if $k > \phi(R)$. Let R be a read that sees $v^{[i,j]}$. If $V^{[k]} \longrightarrow R$, then part (b) of Proposition 3 implies that $k \leq i$, so, by property 1 of ϕ, $k \leq \phi(R)$. By definition of $\xrightarrow{1}$, this implies $V^{[k]} \xrightarrow{1} R$.

Similarly, part (a) of Proposition 3 implies that if $R \longrightarrow V^{[k]}$ then $R \xrightarrow{1} V^{[k]}$. Hence, $\xrightarrow{1}$ is an extension of $\longrightarrow$.

By B0, the relation $\xrightarrow{1}$ is a total ordering on writes, and by definition it totally orders each read with respect to the writes. The next step is to extend $\xrightarrow{1}$ to a total ordering on $\mathcal{S}$, which requires extending it to a total ordering on the set of reads. The restriction of $\xrightarrow{1}$ to the set of reads is just $\longrightarrow$, which is an irreflexive partial ordering. By property 3 of ϕ, we can therefore complete $\xrightarrow{1}$ to a total ordering $\xrightarrow{2}$ of the reads, such that if $\phi(R) < \phi(R')$ then $R \xrightarrow{2} R'$.

Let $\xrightarrow{3}$ be the union of $\xrightarrow{1}$ and $\xrightarrow{2}$. It is clear that for any read and/or write operation executions A and B, either $A \xrightarrow{3} B$ or $B \xrightarrow{3} A$. To show that $\xrightarrow{3}$ is a total ordering—meaning that it is a complete partial ordering, where a partial ordering is transitively closed and irreflexive—it is necessary to show that it is acyclic. Since the restriction of $\xrightarrow{1}$ to the writes is a total ordering and $\xrightarrow{2}$ is a total ordering on the set of reads that extends $\xrightarrow{1}$, any cycle of $\xrightarrow{3}$ must be of the form

$$W_1 \xrightarrow{1} R_1 \xrightarrow{2} \cdots \xrightarrow{2} R_n \xrightarrow{1} W_2 \xrightarrow{1} R_{n+1} \xrightarrow{2} \cdots \xrightarrow{1} W_1$$

where the W_i are writes and the R_j are reads. But such a cycle is impossible because of the following three observations, where R is any read, the first two coming from the definition of $\xrightarrow{1}$ and the second from the definition of $\xrightarrow{2}$:

(a) $V^{[k]} \xrightarrow{1} R$ implies $k \leq \phi(R)$

(b) $R \xrightarrow{1} V^{[k]}$ implies $\phi(R) < k$

(c) $R \xrightarrow{2} R'$ implies $\phi(R) \leq \phi(R')$

Thus, $\xrightarrow{3}$ is a total ordering of $\mathcal{S}$ that extends $\longrightarrow$. Letting $\overset{3}{\dashrightarrow}$ equal $\xrightarrow{3}$ then makes $\langle \mathcal{S}, \xrightarrow{3}, \overset{3}{\dashrightarrow} \rangle$ a system execution. (Axioms A1–A4 follow easily from the fact that $\xrightarrow{3}$ is a total ordering, and A5 follows from the fact that $\xrightarrow{3}$ extends $\longrightarrow$, for which A5 holds.) Thus, $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ trivially implements $\langle \mathcal{S}, \xrightarrow{3}, \overset{3}{\dashrightarrow} \rangle$. To complete the proof of the proposition, it suffices to show that $\langle \mathcal{S}, \xrightarrow{3}, \overset{3}{\dashrightarrow} \rangle$ satisfies B0–B5.

Property B0 is trivial, since it holds for $\longrightarrow$ and $\xrightarrow{3}$ is the same as $\longrightarrow$ on the set of writes. Property B1 is also trivial, since $\xrightarrow{3}$ is a total

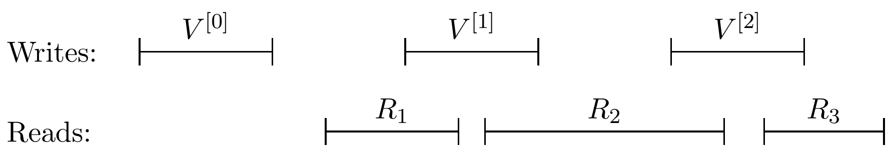

Figure 8: An interesting collection of reads and writes.

ordering. Property B2 follows from the corresponding property for $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$. To prove the remaining properties, observe that the definition of $\xrightarrow{1}$ implies that, in the system execution $\langle \mathcal{S}, \xrightarrow{3}, \overset{3}{\dashrightarrow} \rangle$, any read R sees $v^{[\phi(R),\phi(R)]}$. Properties B3–B5 then follow immediately from the assumption that a read R obtains the value $v^{[\phi(R)]}$. ∎

It was observed above that a regular register can fail to be atomic because two successive reads that overlap the same write could return the new then the old value. Intuitively, Proposition 5 shows that this is the only way a regular register can fail to be atomic. To see this, observe that a function ϕ satisfying properties 1 and 2 of the proposition exists if and only if v is regular. The third property states that two consecutive reads do not obtain out-of-order values.

The exact wording of the proposition is important. One might be tempted to replace the hypothesis with the weaker requirement that v be regular and the following hold:

3′ If $v^{[i,j]} \longrightarrow v^{[i',j']}$ then there exist k and k' with $i \leq k \leq j$ and $i' \leq k' \leq j'$ such that $v^{[i,j]}$ returns the value $v^{[k]}$ and $v^{[i',j']}$ returns the value $v^{[k']}$.

This condition also asserts the same intuitive requirement that two consecutive reads obtain correctly-ordered values, but it does not imply atomicity. As a counterexample, let $v^{[0]} = v^{[2]} = 0$ and $v^{[1]} = 1$, let R_1, R_2, R_3 be the three reads shown in Figure 8, and suppose that R_1 and R_3 return the value 1 while R_2 returns the value 0. (Since each of the reads overlaps a write that changes the value, they all see traces of both values and could return either of them.) The reader (of this paper) can show that this register is regular, but no such ϕ can be constructed; there is no way to interpret these reads and writes as belonging to an atomic register while maintaining the given orderings among the writes and among the reads.

Let us now consider what happens if a global-time model exists. An atomic register is one in which reads and writes do not overlap. Both reads and writes can then be shrunk to a point—that is, reduced to arbitrarily small time intervals within the interval in which they actually occur. For a regular register, it is shown in [5] that reads may be shrunk to a point, so each read overlaps at most one write. However, for a regular register that is not atomic, not all writes can be shrunk to a point.

If two reads cannot overlap the same write, then $v^{[i,j]} \longrightarrow v^{[i',j']}$ implies $j \leq i'$. This implies that any ϕ satisfying conditions 1 and 2 of Proposition 5 also satisfies condition 3. But such a ϕ exists if v is regular, so any regular register trivially implements an atomic one if two reads cannot overlap a single write.

7 Correctness Proofs for the Constructions

7.1 Proof of Constructions 1, 2, and 3

These constructions are all simple, and the correctness proofs are essentially trivial. Formal proofs add no further insight into the constructions, but they do illustrate how the formalism of Part I and the register axioms of the preceding section are applied to actual algorithms. Therefore all the formal details in the proof of Construction 1 are indicated, while the formal proofs for the other two constructions are just briefly sketched.

Recall that in Construction 1, the m-reader register v is implemented by the m single-reader registers v_i. Formally, this construction defines a system, denoted by **S**, that is the set of all system executions consisting of reads and writes of the v_i such that the only operations to these registers are the ones indicated by the readers' and writer's programs. Thus, **S** contains all system executions $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ such that:

- $\mathcal{S}$ consists of reads and writes of the registers v_i.
- Each v_i is written by the same writer and is read only by the i^{th} reader.
- For any i and j: if the write $V_i^{[k]}$ occurs, then the write $V_j^{[k]}$ also occurs and $V_i^{[k-1]} \longrightarrow V_j^{[k]}$.

The third condition expresses the formal semantics of the writer's algorithm, asserting that a write of v is done by writing all the v_i, and that a write of v is completed before the next one is begun.

To say that the v_i are safe or regular means that the system **S** is further restricted to contain only system executions that satisfy B0–B3 or B0–B4, when each v_i is substituted for v in those conditions.

According to Definition 8 of Part I, showing that this construction implements a register v requires constructing a mapping ι from $\mathbf{S}$ to the system $\mathbf{H}$, the latter system consisting of the set of all system executions formed by reads and writes to an m-reader register v. To say that v is safe or regular means that $\mathbf{H}$ contains only system executions satisfying B0–B3 or B0–B4.

In giving the readers' and writer's algorithms, the construction implies that, for each system execution $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ of $\mathbf{S}$, the set $\iota(\mathcal{S})$ of operation executions of $\iota(\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle)$ is the higher-level view of $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ consisting of all writes $V^{[k]}$ of the form $\{V_1^{[k]}, \ldots, V_m^{[k]}\}$, for $V_i^{[k]} \in \mathcal{S}$, and all reads of the form $\{R_i\}$, where $R_i \in \mathcal{S}$ is a read of v_i. (The write $V^{[k]}$ exists in $\iota(\mathcal{S})$ if and only if some, and hence all, $V_i^{[k]}$ exist.) Conditions H1 and H2 of Definition 4 in Part I are obviously satisfied, so this is indeed a higher-level view. To complete the mapping ι, we must define the precedence relations $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ so that $\iota(\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle)$ is defined to be $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$. Proving the correctness of the construction means showing that:

1. $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$ is a system execution. This requires proving that A1–A5 are satisfied.

2. $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ implements $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$. This requires proving that H1–H3 are satisfied.

3. $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$ is in $\mathbf{H}$. This requires proving that B0–B3 or B0–B4 are satisfied.

The precedence relations on $\iota(\mathcal{S})$ are defined to be the "real" ones, with $G \xrightarrow{\mathcal{H}} H$ if and only if G really precedes H. Formally, this means that we let $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ be the induced relations $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$, defined by equations (2) in Section 2 of Part I. It was pointed out in that section that the induced precedence relations make any higher-level view a system execution, so 1 is satisfied. It was already observed that H1 and H2, which are independent of the choice of precedence relations, are satisfied, and H3 is trivially satisfied by the induced precedence relations, so 2 holds. Therefore, it suffices to show that, if B0–B3 or B0–B4 are satisfied for reads and writes of each of the registers v_i in $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$, then they are also satisfied by the register v of $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow} \rangle$.

Properties B0 and B1 for $\langle \iota(\mathcal{S}), \xrightarrow{*}, \overset{*}{\dashrightarrow} \rangle$ follow easily from equations (2) of Part I and the corresponding property for $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$. Property B2 is immediate. The informal proof of B3 is as follows: if a read of v by process i

does not overlap a write (in $\iota(\mathcal{S})$), then the read of v_i does not overlap any write of v_i, so it obtains the correct value. A formal proof is based upon:

X. If a read R_i in $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ sees $v_i^{[k,l]}$, then the corresponding read $\{R_i\}$ in $\langle \iota(\mathcal{S}), \xrightarrow{*}, \overset{*}{\dashrightarrow} \rangle$ sees $v^{[k',l']}$, where $k' \leq k \leq l \leq l'$.

The proof of property X is a straightforward application of (2) of Part I and Definition 9. Property X implies that if B3 or B4 holds for $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$, then it holds for $\langle \iota(\mathcal{S}), \xrightarrow{*}, \overset{*}{\dashrightarrow} \rangle$. This completes the formal proof of Construction 1.

The formal proof of Construction 2 is quite similar. Again, the induced precedence relations are used to turn a higher-level view into a system execution. The proof of Construction 3 is a bit trickier because a write operation to v^* that does not change its value consists only of the read operation to the internal variable x. This means that the induced precedence relation $\overset{*}{\dashrightarrow}$ does not necessarily satisfy B1, so $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$ must be extended to relations $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ for which B1 hold. This is done as follows. For every read-write pair R, W for which neither $R \overset{*}{\dashrightarrow} W$ nor $W \overset{*}{\dashrightarrow} R$ holds, add either one of the relations $R \overset{\mathcal{H}}{\dashrightarrow} W$ or $W \overset{\mathcal{H}}{\dashrightarrow} R$ (it does not matter which), and then add all the extra relations implied by A3, A4, and the transitivity of $\xrightarrow{\mathcal{H}}$. It is then necessary to show that the new precedence relations satisfy A1–A5, the only nontrivial part being the proof that $\xrightarrow{\mathcal{H}}$ is acyclic. Alternatively, one can simply apply Proposition 3 of [5], which asserts the existence of the required precedence relations.

7.2 Proof of Construction 4

The higher-level system execution of reads and writes to v is defined to have the induced precedence relations $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$. As in the above proofs, verifying that this defines an implementation and that B0 and B1 hold is trivial. The only problems are proving B2—namely, showing that the reader must find some v_i equal to one—and proving B4 (which implies B3).

First, the following property is proved:

Y. If a read sees $v^{[l,r]}$ and returns the value μ, then there is some k with $l \leq k \leq r$ such that $v^{[k]} = \mu$.

If B2 holds, then property Y implies B4.

Reasoning about the construction is complicated by the fact that a write of v does not write all the v_j, so the write of v_j that occurs during the kth write of v is not necessarily the k^{th} write of v_j. To overcome this difficulty, new names for the write operations to the v_j are introduced. If v_j is written during the execution of $V^{[k]}$, then $W_j^{[k]}$ denotes that write of v_j; otherwise, $W_j^{[k]}$ is undefined. Thus, every write $V_j^{[l]}$ of v_j is also named $W_j^{[l']}$ for some $l' \geq l$. A read of v_j is said to see $w_j^{[l',r']}$ if it sees $v_j^{[l,r]}$ and the writes $W_j^{[l']}$ and $W_j^{[r']}$ are the same writes as $V_j^{[l]}$ and $V_j^{[r]}$, respectively. Note that, because the writer's algorithm writes from "right to left", $W_1^{[k]}$ exists for all k and, if $W_i^{[k]}$ exists, then so do all the $W_j^{[k]}$ with $j < i$.

Let R be a read that returns the value μ, and let μ be the i^{th} value, so R consists of the sequence of reads $R_1 \longrightarrow \cdots \longrightarrow R_i$, where each R_j is a read of v_j. All the R_j return the value 0 except R_i, which returns the value 1. Let R see $v^{[l,r]}$ and let each R_j see $w_j^{[l(j),r(j)]}$. By regularity of v_j, there is some $k(j)$ with $l(j) \leq k(j) \leq r(j)$ such that $W_i^{[k(i)]}$ writes a 1 and $W_j^{[k(j)]}$ writes a 0 for $1 \leq j < i$. Thus, $v^{[k(i)]}$ is the value read by R, so it suffices to show that $l \leq k(i) \leq r$.

Definition 9 applied to the read R_i of v implies $W_i^{[r(i)]} \dashrightarrow R_i$, which, by equation (2) of Part I, implies $V^{[r(i)]} \overset{*}{\dashrightarrow} R$. This in turn implies $r(i) \leq r$, so $k(i) \leq r$.

For any p with $p \leq l$, Definition 9 implies that $R \overset{*}{\not\dashrightarrow} V^{[p]}$, which implies that $R_1 \not\dashrightarrow W_1^{[p]}$, which in turn implies that $p \leq l(1)$. Hence, letting $p = l$, we have $l \leq l(1)$.[10] Since $l(j) \leq k(j)$, it suffices to prove that $k(j) \leq l(j+1)$ for $1 \leq j < i$.

Since $k(j) \leq r(j)$, Definition 9 implies that $W_j^{[k(j)]} \dashrightarrow R_j$. Because $W_j^{[k(j)]}$ writes a zero, $W_{j+1}^{[k(j)]}$ exists, and we have

$$W_{j+1}^{[k(j)]} \longrightarrow W_j^{[k(j)]} \dashrightarrow R_j \longrightarrow R_{j+1}$$

where the two $\longrightarrow$ relations are implied by the order in which writing and reading of the individual v_j are performed. By A4, this implies that $W_{j+1}^{[k(j)]} \longrightarrow R_{j+1}$, which, by A2, implies $R_{j+1} \not\dashrightarrow W_{j+1}^{[k(j)]}$. By Definition 9, this implies that $k(j) \leq l(j+1)$, completing the proof of property Y.

To complete the proof of the construction, it suffices to prove that every read does return a value. Let R and the values $l(j)$, $k(j)$, and $r(j)$ be as

[10]Note that the same argument does not prove that $l \leq l(i)$ because $W_i^{[p]}$ does not necessarily exist.

above, except let $i = n$ and drop the assumption that R_i obtains the value 1. To prove B2, it is necessary to prove that R_n does obtain the value 1.

The same argument used above shows that, if R_j obtains a zero, then that zero was written by some write $W_j^{[k(j)]}$, which implies that $W_{j+1}^{[k(j)]}$ exists and $k(j) \leq l(j+1)$. Since R_n obtains the value written by $W_n^{[k(n)]}$, it must obtain a 1 unless $k(n) = 0$ and the initial value is not the nth one. Suppose the initial value $v^{[0]}$ is the pth value, encoded with $v_p = 1$, $p < n$. Since R_p obtains the value 0, we must have $k(p) > 0$, which implies that $k(n) > 0$, so R_n obtains the value 1. This completes the proof of the construction.

7.3 Proof of Construction 5

This construction defines a set $\mathcal{H}$, consisting of reads and writes of v^*, that is a higher-level view of a system execution $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ whose operation executions are reads and writes of the two shared registers v and c. As usual, $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$ denote the induced precedence relations on $\mathcal{S}$ that are defined by (2) of Part I.

In this construction, the write $V^{*[k+1]}$ of v^*, for $k \geq 0$, is implemented by the sequence

$$RC_k \longrightarrow V^{[3k+1]} \longrightarrow V^{[3k+2]} \longrightarrow V^{[3k+3]} \qquad (3)$$

where $num(v^{[3k+i]}) = i$ and RC_k is a read of c that obtains the value $\neg col(v^{[3k+i]})$, the colors $col(v^{[3k+1]})$ being the same for the three values of i. (Recall that $V^{[p]}$ is the pth write of v and $v^{[p]}$ is the value it writes.) The initial write $V^{*[0]}$ of v^* is just the initial write $V^{[0]}$ of v.

Since there is only one reader, the reads of v^* are totally ordered by $\xrightarrow{*}$. The jth read R_j^* of v^* consists of the sequence $RV_j \longrightarrow C^{[j]}$, where RV_j is the jth read of v and $C^{[j]}$ is the jth write of c.

The proof of correctness is based upon Proposition 5. Letting $\phi(j)$ denote $\phi(R_j^*)$, to apply that proposition, it suffices to choose the $\phi(j)$ such that the following three properties hold:

1. If R_j^* sees $v^{*[l,r]}$ then $l \leq \phi(j) \leq r$.

2. R_j^* returns the value $v^{*[\phi(j)]}$.

3. If $j' < j$ then $\phi(j') \leq \phi(j)$.

Intuitively, the existence of such a function ϕ means we can pretend that the read R_j^* occurred after the $\phi(j)^{\text{th}}$ write and before the $\phi(j)+1^{\text{st}}$ write of v^*.

To construct such a ϕ, a function ψ is first defined such that RV_j returns the value $v^{[\psi(j)]}$ and, if RV_j sees $v^{[l,r]}$, then $l \leq \psi(j) \leq r$. Since v is regular, such a ψ exists. From part (c) of Proposition 3, we have:

$$j' < j \text{ implies } \psi(j') \leq \psi(j) + 1 \tag{4}$$

We define $\phi(j)$ as follows. If $\psi(j) = 3k + i$, with $1 \leq i \leq 3$, then $\phi(j)$ equals k if R_j^* returns the value $old(rv)$ (by executing the innermost **else** clause of the reader's algorithm) and it equals $k+1$ if R_j^* returns the value $new(rv)$. We must now prove that ϕ satisfies properties 1–3.

By Proposition 4, to prove property 1 it suffices to prove:

$$V^{*[\phi(j)]} \overset{*}{\dashrightarrow} R_j^* \overset{*}{\dashrightarrow} V^{*[\phi(j)+1]} \tag{5}$$

Proposition 4 implies that

$$V^{[\psi(j)]} \dashrightarrow RV_j \dashrightarrow V^{[\psi(j)+1]} \tag{6}$$

If $\psi(j) = 3k+3$, then $V^{[\psi(j)]}$ is part of $V^{*[k+1]}$ and $V^{[\psi(j)+1]}$ is part of $V^{*[k+2]}$, so (6) and the definition of $\overset{*}{\dashrightarrow}$ imply

$$V^{*[k+1]} \overset{*}{\dashrightarrow} R_j^* \overset{*}{\dashrightarrow} V^{*[k+2]}$$

But $\psi(j) = 3k+3$ implies that R_j^* obtains $num(rv) = 3$ and therefore returns $new(rv)$, so, by definition of ϕ, $\phi(j) = k+1$, which proves (5).

If $\psi(j) = 3k+i$ with $1 \leq i \leq 3$, then $V^{[\psi(j)]}$ and $V^{[\psi(j)+1]}$ are both part of $V^{*[k+1]}$, so (6) and the definition of $\overset{*}{\dashrightarrow}$ imply

$$V^{*[k+1]} \overset{*}{\dashrightarrow} R_j^* \overset{*}{\dashrightarrow} V^{*[k+1]}$$

Since $V^{*[k]} \xrightarrow{*} V^{*[k+1]} \xrightarrow{*} V^{*[k+2]}$, (5) follows from Axiom A3 when $\phi(j)$ equals either k or $k+1$, which, by definition of ϕ, are the only two possibilities. This finishes the proof of (5), which proves property 1.

Property 2 follows immediately from the definition of ϕ and the observation that if $1 \leq i \leq 3$, then $v^{*[k]} = old(v^{[3k+i]})$ and $v^{*[k+1]} = new(v^{[3k+i]})$.

To prove property 3, it suffices to show that, for every j, $\phi(j-1) \leq \phi(j)$. By (4), $\psi(j-1) \leq \psi(j)+1$. It therefore follows from the definition of ϕ that there are only two situations in which $\phi(j-1)$ could be greater than $\phi(j)$:

(a) $\psi(j) = 3k + i$, $1 \le i \le 3$, and R^*_j returns $\mathit{old}(rv)$, and
$\psi(j-1) = 3k + i'$, $1 \le i' \le 3$, and R^*_{j-1} returns $\mathit{new}(rv)$.

(b) $\psi(j) = 3k$, $\psi(j-1) = 3k+1$, and R^*_{j-1} returns $\mathit{new}(rv)$.

We first show that case (a) is impossible. Since $\psi(j-1) \le \psi(j) + 1$, we have $i' \le i + 1$. However, i is the value of $\mathit{num}(rv)$ obtained by R^*_j, while i' is the value of $\mathit{num}(rv)$ obtained by R^*_{j-1} and hence the value of $\mathit{num}(rv')$ during the execution of R^*_j (after it executes the first assignment statement). Therefore, when executing R^*_j, the reader finds nuret true (because R^*_{j-1} returned $\mathit{new}(rv)$), $\mathit{col}(rv) = \mathit{col}(rv')$ (because both R^*_j and R^*_{j-1} obtained values written by the same write $V^{*[k+1]}$), and $\mathit{num}(rv) \ge \mathit{num}(rv') - 1$ (because $i' \le i+1$). Hence R^*_j must return $\mathit{new}(rv)$, so case (a) is impossible.

Finally, we show the impossibility of case (b). This is the most difficult part of the proof, and essentially involves proving the assertion made in Section 5 that, if a read obtains the value $(\mu, \nu, 1, \kappa)$ and returns the value ν, then it and a preceding read both overlap a write of the value $(\mu, \nu, 2, \kappa)$.

Examination of the reader's algorithm reveals that for case (b) to occur, there must exist reads $R^*_{j_3}$ and $R^*_{j_2}$ such that (i) $j_3 < j_2 < j - 1$, (ii) each $R^*_{j_i}$ obtains a value of rv with $\mathit{num}(rv) = i$ and $\mathit{col}(rv)$ equal to the value of $\mathit{col}(rv)$ obtained by R^*_{j-1}, and (iii) every read between $R^*_{j_3}$ and R^*_{j-1} also obtains the same value of $\mathit{col}(rv)$ as R^*_{j-1}. For notational convenience, let $j_1 = j - 1$ and let κ denote the value of $\mathit{col}(rv)$ obtained by the reads $R^*_{j_i}$. We then have:

$$RV_{j_3} \longrightarrow C^{[j_3]} \longrightarrow RV_{j_2} \longrightarrow C^{[j_2]} \longrightarrow RV_{j_1} \longrightarrow C^{[j_1]} \tag{7}$$

$$j_3 \le j \le j_1 \text{ implies } c^{[j]} = \kappa \tag{8}$$

Since $R^*_{j_i}$ obtains $\mathit{num}(rv) = i$, $\psi(j_i)$ equals $3k_i + i$ for some k_i. Since $R^*_{j_i}$ obtains $\mathit{col}(rv) = \kappa$, RC_{k_i} reads the value $\neg\kappa$. (Remember that RC_k is the read of c that is part of the write $V^{*[k+1]}$.

Since $\psi(j_i) = 3k_i + i$, substituting j_i for j in (6) yields

$$V^{[3k_i+i]} \dashrightarrow RV_{j_i} \tag{9}$$

$$RV_{j_i} \dashrightarrow V^{[3k_i+i+1]} \tag{10}$$

We show now that $k_1 = k_2$, which shows that $R^*_{j_2}$ and $R^*_{j_1}$ overlap the same write of v^*. The proof is by contradiction. First, assume that $k_2 > k_1$. This implies that $V^{[3k_1+2]} \longrightarrow V^{[3k_2+2]}$, which, with (7) and (10), yields

$$RV_{j_2} \longrightarrow RV_{j_1} \dashrightarrow V^{[3k_1+2]} \longrightarrow V^{[3k_2+2]}$$

Applying Axiom A4, we get $RV_{j_2} \longrightarrow V^{[3k_2+2]}$, and, by Axiom A2, this contradicts (9), so we must have $k_2 \leq k_1$.

Next, assume that $k_2 < k_1$. This implies that $V^{[k_2+3]} \longrightarrow RC_{k_1}$. Combining this with (7) and (10) gives

$$C^{[j_3]} \longrightarrow RV_{j_2} \dashrightarrow V^{[k_2+3]} \longrightarrow RC_{k_1}$$

and Axiom A4 implies

$$C^{[j_3]} \longrightarrow RC_{k_1} \tag{11}$$

Let l and r be integers such that RC_{k_1} sees $c^{[l,r]}$. By part (b) of Proposition 3, (11) implies that $j_3 \leq l$. Since RC_{k_1} obtains the value $\neg\kappa$, (8) and the regularity of c (Axiom B4) imply that $r > j_1$. Part (a) of Proposition 4 (substituting $j_1 + 1$ for k and r for j) then implies $C^{[j_1+1]} \dashrightarrow RC_{k_1}$. Since $C^{[j_1+1]}$ is part of a later read operation execution than is RV_{j_1}, we have $RV_{j_1} \longrightarrow C^{[j_1+1]}$. Combining these two relations with (3) gives

$$RV_{j_1} \longrightarrow C^{[j_1+1]} \dashrightarrow RC_{k_1} \longrightarrow V^{[3k_1+1]}$$

which by A4 implies $RV_{j_1} \longrightarrow V^{[3k_1+1]}$. Axiom A2 and (9) imply that this is impossible, so we have the contradiction that completes the proof that $k_1 = k_2$.

Returning to (b), recall that $j_1 = j - 1$ and $k_1 = k$. We have $\psi(j) = 3k$, $\psi(j_2) = 3k_2 + 2 = 3k + 2$, and $j_2 < j - 1 < j$, which contradicts (4). Hence, this shows that (b) is impossible, which completes the proof of property 3, completing the proof of correctness of the construction.

8 Conclusion

I have defined three classes of shared registers for asynchronous interprocess communication and have provided algorithms for implementing stronger classes in terms of weaker ones. For single-writer registers, the only unsolved problem is implementing a multireader atomic register. A solution probably exists, but it undoubtedly requires that a reader communicate with all other readers as well as with the writer. Also, more efficient implementations than Constructions 4 and 5 probably exist. For multivalued registers, Peterson's algorithm [14] combined with Construction 5 provides a more efficient implementation of a regular register than Construction 4, and a more efficient implementation of a single-reader atomic register than Construction 5. However, in this solution, Construction 4 is still needed to implement the regular register used in Construction 5.

The only closely related work that I know of is that of Misra [13]. Misra's main result is a generalization of a restricted version of Proposition 5 of Section 6. It generalizes the proposition to multiple writers, but assumes a global-time model rather than using the more general formalism of Part I.

I have not addressed the question of multiwriter shared registers. It is not clear what assumptions one should make about the effect of overlapping writes. The one case that is straightforward is that of an atomic multiwriter register—the kind of register traditionally assumed in shared-variable concurrent programs. This raises the problem of implementing a multiwriter atomic register from single-writer ones. An unpublished algorithm of Bard Bloom implements a two-writer atomic register using single-writer atomic registers.

The definitions and proofs have all employed the general formalism developed in Part I. Instead of the more traditional approach of considering starting and stopping times of the operation executions, this formalism is based upon two abstract precedence relations satisfying Axioms A1–A5. These axioms embody the fundamental properties of temporal relations among operation executions that are needed to analyze concurrent algorithms.

Acknowledgements

The algorithms described here were derived many years ago, although they have never before appeared in print. Carel Scholten provided much of the intellectual stimulation that led to their discovery. Jay Misra is largely responsible for persuading me to finally publish them, and Fred Schneider helped make the current version less unreadable than previous ones.

References

[1] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–222, June 1981.

[2] W. Brauer, editor. *Net Theory and Applications.* Springer-Verlag, Berlin, 1980.

[3] P. J. Courtois, F. Heymans, and David L. Parnas. Concurrent control with “readers” and “writers”. *Communications of the ACM*, 14(10):190–199, October 1971.

[4] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[5] Leslie Lamport. *Interprocess Communication.* Technical Report, SRI International, March 1985.

[6] Leslie Lamport. The mutual exclusion problem. To appear in *JACM*.

[7] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[8] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] Leslie Lamport. 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, January 1985.

[10] Peter E. Lauer, Michael W. Shields, and Eike Best. *Formal Theory of the Basic COSY Notation.* Technical Report TR143, Computing Laboratory, University of Newcastle upon Tyne, 1979.

[11] A. Mazurkiewicz. *Semantics of Concurrent Systems: A Modular Fixed Point Trace Approach.* Technical Report 84–19, Institute of Applied Mathematics and Computer Science, University of Leiden, 1984.

[12] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, Berlin, 1980.

[13] J. Misra. Axioms for memory access in asynchronous hardware systems. 1984. To appear in *ACM Transactions on Programming Languages and Systems*.

[14] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[15] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th Symposium on the Foundations of Computer Science*, ACM, November 1977.

[16] Glynn Winskel. *Events in Computation.* PhD thesis, Edinburgh University, 1980.