Arbitration-Free Synchronization

Leslie Lamport

22 March 2002
(minor corrections made on 30 September 2002)

To appear in the PODC Jubilee Issue of
Distributed Computing.

Abstract

Implementing traditional forms of multiprocess synchronization requires a hardware arbiter. Here, we consider what kind of synchronization is achievable without arbitration. Several kinds of simple arbiter-free registers are defined and shown to have equal power, and the class of synchronization problems solvable with such registers is characterized. More powerful forms of arbiter-free communication primitives are described. However, the problem of characterizing the most general form of arbiter-free synchronization remains unsolved.

Contents

1 Introduction

In classic multiprocess synchronization algorithms, processes communicate by reading and writing shared registers. The most primitive interprocess communication mechanism that has been considered seems to be the one-reader *safe* one-bit register, introduced in [11]. This register can be written by one process and read by one other process, a read obtaining the "correct" value if it does not overlap a write. Such registers can be used to solve any kind of synchronization problem, though not necessarily in a wait-free manner [7]. However, implementing a read/write register requires an arbiter—a device that makes a discrete decision based on a continuous range of inputs. It is impossible to build an arbiter that always decides within a bounded length of time [2]. Hence, synchronization using a shared register may take an unbounded length of time. We study here what kind of synchronization can be achieved in bounded time, which means without using an arbiter.

Without an arbiter, we can implement an interprocess communication mechanism called a wait/signal register, in which one process waits for a signal from another process. We describe five types of wait/signal registers and show that they are all equivalent, in the sense that each can be used to implement the others. We then present our main result, that the synchronization achievable by finite-state deterministic processes communicating with wait/signal registers is essentially that described by marked graphs [3], a special class of Petri nets [18] that generalize producer/consumer synchronization.

At least one special case of our result has been known for some time. It is well-known among experts on self-timed circuits [16] that producer/consumer synchronization can be implemented without an arbiter. (This is the basic form of synchronization achieved by Sutherland's micropipelines [19].) We do not know if the full generalization to marked graphs was previously known. While arbiter-free self-timed circuits have been considered [20, 21], we know of no characterization of the interprocess synchronization that they can effect.

Wait/signal registers are not the only synchronization mechanism implementable without an arbiter. We can also implement a weak read/write register, which is one that may not be accessed concurrently by different processes. Adding weak read/write registers does not significantly alter the class of solvable synchronization problems unless we also allow processes to make nondeterministic choices. Implementing nondeterminism seems to require an arbiter. However, we can view a process's nondeterministic choice as representing input to an arbiter-free system from an environment that may contain arbiters. We briefly describe a class of synchronization problems that can be solved by nondeterministic processes with wait/signal and weak read/write registers. However, we do not know if this is the most general class of problems solvable with such a system.

We can enhance wait/signal registers without introducing arbitration by allowing a process to wait for a signal from any of a set of processes. We give an example to show that this enhancement extends the power of wait/signal registers, but we do not know how to characterize the synchronization it allows.

In the next section, we review some concepts for describing nonatomic operations and define what we mean by multiprocess synchronization. Section 3 discusses arbitration and shows that some classic synchronization problems require an arbiter. Section 4 defines several types of wait/signal register and proves their equivalence. It also shows how to implement alternation with wait/signal registers. Section 5 reviews marked graphs and defines a subclass we call process marked graphs. Section 6 shows that process marked graphs essentially describe the synchronization achievable by finite-state processes with wait/signal registers. Section 7 briefly discusses other synchronization primitives, and we conclude with an overview of what we have done and what remains undone.

The basic question that we ask is, what kind of multiprocess synchronization can be achieved without arbitration? Our attempt to answer it is far from satisfactory. We have found that just asking the question precisely is difficult and requires a somewhat arcane formalism. We have only a partial answer—namely, a characterization of what can be implemented with wait/signal registers, which are just one class of arbiter-free synchronization primitive. And even that answer is rather complicated. Much of our exposition is informal; we have been formal only where we feel that our formalism is simple and compelling. Still, we feel that the question is an interesting one, and we hope that our results will stimulate further progress towards answering it. This work is still in the realm of pure theory; we know of no practical application of arbiter-free multiprocess synchronization.

2 Systems and Synchronization

Before discussing arbiter-free multiprocess synchronization, we have to understand what multiprocess synchronization is. In this section, we define what a multiprocess system is, what a synchronization problem is, and what it means for a multiprocess system to solve a synchronization problem.

The standard formalisms for describing systems, such as temporal logic [14] and process algebra [17], describe atomic actions. The fundamental problem of multiprocess synchronization is that operations are not inherently atomic and can be executed concurrently by different processes. We therefore use a formalism that was introduced for describing nonatomic operations [10, 12]. However, our exposition is rather informal.

2.1 System Executions

A system is described by a set of *system executions*, each one representing a possible (legal) execution of the system. A system execution is a structure consisting of a set H of *operation executions* and two relations $\longrightarrow$ and $\dashrightarrow$ on H. With each operation execution in H we associate an operation. If O is the operation associated with an operation execution A, we say that A is an execution of operation O. We usually write an operation execution as $O^{[i]}$, where O is its operation and i is an integer that distinguishes this operation execution from other executions of O. Usually, $O^{[i]}$ is the i^{th} execution of operation O.

For operation executions A and B in a system execution, we say that A *precedes* B if $A \longrightarrow B$, that A *can affect* B if $A \dashrightarrow B$, and that A and B are *concurrent* if $A \not\longrightarrow B$ and $B \not\longrightarrow A$.

To motivate the relations $\longrightarrow$ and $\dashrightarrow$ and their properties, we can think of an operation execution as comprising a set of events. These events might be discrete atomic events in some lower-level model of the system, or they might be the points in a continuous region of space-time at which the operation execution is performed by some physical device. We assume an irreflexive partial order on the set of all events, where $a \to b$ means that event a happens before event b [8]. An operation execution A precedes an operation execution B iff every event of A precedes every event of B:

$$(1) \quad A \longrightarrow B \;\equiv\; \forall\, a \in A,\, b \in B \,:\, a \to b$$

Operation execution A can affect operation execution B iff A can influence the result of B—for example, if A writes a value to a register and B reads at least part of that value. For this to be the case, some event of A has to precede some event of B:[1]

$$(2) \quad A \dashrightarrow B \text{ implies } \exists\, a \in A,\, b \in B \,:\, a \to b$$

[1] In previous work [10, 12], (2) was taken to be an equivalence rather than an implication. Here, we are taking $\dashrightarrow$ to be an actual causality relation, not just a temporal relation in which causality is possible.

The following rules are simple consequences of (1) and (2) and of the assumption that $\rightarrow$ is an irreflexive partial order:

R1. $\longrightarrow$ is an irreflexive partial order.

R2. $A \longrightarrow B \dashrightarrow C \longrightarrow D$ implies $A \longrightarrow D$, for all $A, B, C, D \in H$.

We have used (1) and (2) to motivate rules R1 and R2 in terms of events. However, events are not part of the formalism, and we take R1 and R2 to be axioms. A formal characterization of system executions requires additional axioms, but we will not need them here.

A *multiprocess* system is one in which we assign one of a finite number of processes to each operation. If A is an execution of operation O to which we assign process p, we say that *p performs A*. We assume that an algorithm can enforce precedence relations between operation executions performed by the same process. However, we allow a process to execute some operations concurrently, so we don't assume that all the operation executions of a process are totally ordered.

As we shall see, the purpose of a synchronization algorithm is to enforce precedence relations among operation executions. We assume that a process's algorithm enforces intraprocess precedence relations. Interprocess communication primitives guarantee only causality relations between operation executions of different processes. For example, if a read R obtains the value written by a write W, we can conclude that $W \dashrightarrow R$, not that $W \longrightarrow R$. Interprocess precedence relations are obtained with rule R2 from intraprocess precedence relations and interprocess causality relations.

An operation execution may or may not terminate. If A is a nonterminating operation execution, then $A \not\longrightarrow B$ holds for all operation executions B. The converse is not necessarily true.

2.2 A Safe Read/Write Register

As an example of how system executions describe systems, we consider a system containing a one-reader, one-writer one-bit safe register [13]. A writer process performs a (finite or infinite) sequence $W(w_1)^{[1]} \longrightarrow W(w_2)^{[2]} \longrightarrow \cdots$ of operation executions, where $W(w_i)^{[i]}$ represents the writing of the value w_i, which is 0 or 1, to the register. A reader process likewise performs a sequence $R(r_1)^{[1]} \longrightarrow R(r_2)^{[2]} \longrightarrow \cdots$ of operation executions, $R(r_i)^{[i]}$ representing a read that obtains the value r_i. The relation $W(w_i)^{[i]} \dashrightarrow R(r_j)^{[j]}$ means that the j^{th} read can "see traces of" the i^{th} write. The register is specified by the requirement that, for every execution of the system, the following conditions hold for all i and j.

S1. If $W(w_i)^{[i]} \longrightarrow R(r_j)^{[j]}$ and either $R(r_j)^{[j]} \longrightarrow W(w_{i+1})^{[i+1]}$ or $W(w_i)^{[i]}$ is the last write, then $r_j = w_i$; and if $R(r_j)^{[j]} \longrightarrow W(w_1)^{[1]}$, then $r_j = 0$. (A read not concurrent with any write obtains the most recently written value, where the initial value is assumed to be 0.)

S2. r_j equals 0 or 1. (Every read returns a legal value.)

S3. If $W(w_i)^{[i]}$ and $R(r_j)^{[j]}$ are concurrent, then $W(w_i)^{[i]} \dashrightarrow R(r_j)^{[j]}$. (If a read and a write execution occur and neither precedes the other, then the read might see traces of the write.)

2.3 The Muller C-Element

As another example of a system description, we consider a system containing a Muller C-element, a basic component of self-timed circuits [16]. An n-input C-element is a device with n input lines and one output line that produces an output signal after receiving an input signal on every input line. After a signal appears on an input line, the next signal on that line must not occur until after the C-element produces an output.

An execution of a system with a 2-input C-element contains the following operation executions:[2]

$$\begin{tikzcd}
\text{Input 1 Operation Executions:} & I_1^{[1]} \arrow[r] \arrow[d, dashed] \arrow[dr] & I_1^{[2]} \arrow[r] \arrow[d, dashed] \arrow[dr] & I_1^{[3]} \arrow[r] \arrow[d, dashed] \arrow[dr] & \cdots \\
\text{C-Element Operation Executions:} & O^{[1]} \arrow[r] & O^{[2]} \arrow[r] & O^{[3]} \arrow[r] & \cdots \\
\text{Input 2 Operation Executions:} & I_2^{[1]} \arrow[r] \arrow[u, dashed] \arrow[ur] & I_2^{[2]} \arrow[r] \arrow[u, dashed] \arrow[ur] & I_2^{[3]} \arrow[r] \arrow[u, dashed] \arrow[ur] & \cdots
\end{tikzcd}$$

The solid arrows are assumed—the horizontal ones are the ordering of operation executions within a process, the diagonal ones must be guaranteed by the system. The dashed arrows and termination of the $O^{[i]}$ operations are ensured by the C-element.

In general, an n-input C-element system contains input operation executions $I_j^{[i]}$ for each input j and output operation executions $O^{[i]}$. We assume that (i) the output operation executions $O^{[i]}$ are totally ordered, (ii) for each input process j, the operation executions $I_j^{[i]}$ are totally ordered, and (iii) for each i, operation execution $O^{[i+1]}$ occurs only if $I_j^{[i]}$ occurs, for every j, and that $I_j^{[i]} \longrightarrow O^{[i+1]}$. The C-element then guarantees that (i) the operation

[2] The C-element is traditionally described in terms of circuit behavior, not operation executions. We have translated that description into our formalism.

executions $I_j^{[i]}$ terminate, (ii) an operation execution $O^{[i]}$ terminates iff $I_j^{[i]}$ occurs, for all inputs j, and (iii) $I_j^{[i]} \dashrightarrow O^{[i]}$ for each j. The specification should actually assert that these guarantees hold for a particular i if the assumptions hold for all smaller values of i, but we won't bother stating this precisely.

2.4 Implementation

When implementing a higher-level system by a lower-level one, we must explain how to interpret an execution of the lower-level one as an execution of the higher-level one. This requires explaining how a set of operation executions in an execution of the lower-level system is interpreted as a single operation execution of the higher-level system. The $\rightarrow$ and $\dashrightarrow$ relations of the higher-level interpretation are defined in terms of the $\rightarrow$ and $\dashrightarrow$ relations of the lower-level system executions as follows. If $\mathcal{A}$ and $\mathcal{B}$ are sets of lower-level operation executions that are considered to be individual operation executions of the higher-level system, then:

$$(3) \quad \mathcal{A} \rightarrow \mathcal{B} \;\triangleq\; \forall\, A \in \mathcal{A},\, B \in \mathcal{B} \,:\, A \rightarrow B$$

$$(4) \quad \mathcal{A} \dashrightarrow \mathcal{B} \;\triangleq\; \exists\, A \in \mathcal{A},\, B \in \mathcal{B} \,:\, A \dashrightarrow B$$

It is easy to check that R1 and R2 are satisfied by the high-level interpretation if they are satisfied by the lower-level system execution.

A set $\mathcal{A}$ of operation executions of the lower-level system represents a terminating operation of the higher-level system iff $\mathcal{A}$ is finite and all its elements are terminating operation executions.

2.5 Synchronization Problems

We define a synchronization problem to be a requirement on the $\rightarrow$ relations among a collection of operation executions. Solving the problem means adding operation executions that perform the synchronization needed to guarantee that every system execution satisfies the required $\rightarrow$ relations.

The best-known synchronization problem is mutual exclusion. In this problem, each process executes a sequence of critical-section operations, and we require that all critical-section executions be totally ordered by $\rightarrow$. In other words, no two processes may execute their critical sections concurrently.

Another important synchronization problem is the *alternation* problem. In this problem a "producer" process performs a (possibly infinite) sequence

$P^{[1]} \longrightarrow P^{[2]} \longrightarrow \cdots$ of operation executions and a "consumer" process performs a sequence $C^{[1]} \longrightarrow C^{[2]} \longrightarrow \cdots$ of operation executions. We require that the P and C operation executions alternate:

$$P^{[1]} \longrightarrow C^{[1]} \longrightarrow P^{[2]} \longrightarrow C^{[2]} \longrightarrow \cdots$$

More precisely, we require that the following two conditions hold, for any $i \geq 1$:

A1. If execution $C^{[i]}$ occurs, then so does $P^{[i]}$, and $P^{[i]} \longrightarrow C^{[i]}$.

A2. If $P^{[i+1]}$ occurs, then so does $C^{[i]}$, and $C^{[i]} \longrightarrow P^{[i+1]}$.

The restrictions on the $\longrightarrow$ relations posed by a synchronization problem are safety requirements. Synchronization problems also have liveness requirements. For example, we usually require a solution of the mutual exclusion problem to be "deadlock-free". This means that a process wanting to execute a critical-section operation must eventually do so, if all other processes stop executing critical-section operations. The "starvation-free" version of the mutual exclusion problem strengthens this by requiring that any process that wants to execute a critical-section operation must eventually do so, even if other processes keep executing critical-section operations.

The alternation problem's safety requirement implies that of the mutual exclusion problem, since it implies that all the $P^{[i]}$ and $C^{[i]}$ operations are totally ordered. If we ignored liveness requirements, we could therefore consider a solution to the alternation problem to solve the two-process mutual exclusion problem, with the $P^{[i]}$ and $C^{[i]}$ being the critical-section operation executions. However, as we explain below, the two problems are fundamentally very different.

To formalize the liveness part of a synchronization problem, we would have to introduce *request* operations. For example, in the mutual exclusion problem, deadlock freedom means that, in any execution containing only a finite number of critical-section operation executions, each request is followed by a critical-section operation. Starvation freedom means that this condition holds for the operation executions of each process. For the class of synchronization problems that primarily concern us, liveness poses no serious problem, and little is gained by treating it formally. So, we will content ourselves with an informal treatment of liveness.

2.6 Operation Execution Graphs

We now generalize the alternation problem to the *n-buffer producer/consumer problem*, for any positive integer n. The alternation problem is the

$n = 1$ case. As in the alternation problem, the producer and consumer each perform an infinite sequence of operation executions $P^{[i]}$ and $C^{[i]}$, respectively, for all positive integers i. We require that condition A1 above and the following condition hold for all $i \geq 1$:

A2′. If $P^{[i+n]}$ occurs, then so does $C^{[i]}$, and $C^{[i]} \longrightarrow P^{[i+n]}$.

Think of the producer communicating a sequence of values to the consumer using n buffers. Operation execution $P^{[i]}$ writes the i^{th} value in buffer $i \bmod n$, and $C^{[i]}$ reads that value from the buffer. We require that the i^{th} value be written before it can be read ($P^{[i]} \longrightarrow C^{[i]}$), and that it be read before the next value is written into the same buffer ($C^{[i]} \longrightarrow P^{[i+n]}$). Here is an illustration for $n = 3$:

(5)
$$
\begin{tikzcd}
\text{Producer:} & P^{[1]} \arrow[r] \arrow[d] & P^{[2]} \arrow[r] \arrow[d] & P^{[3]} \arrow[r] \arrow[d] & P^{[4]} \arrow[r] \arrow[d] & P^{[5]} \arrow[r] \arrow[d] & \cdots \\
\text{Consumer:} & C^{[1]} \arrow[r] \arrow[urrr] & C^{[2]} \arrow[r] \arrow[urrr] & C^{[3]} \arrow[r] \arrow[urrr] & C^{[4]} \arrow[r] \arrow[urr, dash] & C^{[5]} \arrow[r] \arrow[ur, dash] & \cdots
\end{tikzcd}
$$

This diagram is an example of what we call an *operation execution graph*, which is an acyclic directed graph whose nodes are operation executions such that (i) each node has only a finite number of incoming and outgoing edges, (ii) any path has a first node, and (iii) for each process, there is a path whose nodes are the set of operation executions of that process. (Condition (ii) means that there is no infinite sequence $n_1, n_2, \ldots$ of nodes such that there is an edge from n_{i+1} to n_i, for each i.)

An operation execution graph E defines a synchronization problem, where each edge in the graph represents a required $\longrightarrow$ relation. The safety part of the problem requires that a system execution be a prefix of the complete operation execution graph. More precisely, a set OE of operation executions with a relation $\longrightarrow$ satisfies the safety requirement defined by the execution graph E iff (i) OE is a subset of the nodes of E and (ii) if there is a path from node n to node p in E and the operation execution p is in OE, then the operation execution n is in OE and $n \longrightarrow p$. Note that this synchronization problem requires the operations of a single process to be totally ordered. However, we allow concurrent executions of the operations used to implement the required synchronization.

For an operation execution graph, there are three natural choices for the liveness property that we require. If we assume that the operation executions are generated by the system and are not issued in response to requests, then these three liveness properties are:

None A system execution may be any "prefix" of the operation execution graph.

Deadlock Freedom If the graph is infinite, then any system execution must contain an infinite subset of the graph's operation executions.

Process Fairness Any system execution must contain all of the graph's operation executions.

It is not hard to define the analogous conditions if operations are executed in response to external requests.

In the n-buffer producer/consumer problem, the producer must perform at least as many operation executions as the consumer, and it can perform at most n more. Hence, deadlock freedom implies process fairness. A trivial example in which the two liveness conditions are not equivalent is provided by an operation execution graph in which each process executes an infinite sequence of operations, but there are no interprocess edges.

Since the $\longrightarrow$ relation on a system execution is transitive, the synchronization problem defined by an operation execution graph depends only on the connectivity properties of the graph. Adding an edge from node n to node p does not change the problem if there is already a path from n to p in the graph. We define a 1-1 function κ from the nodes of an operation execution graph E to those of an operation execution graph E' to be an *equivalence* iff (i) every node n of E has the same associated operation as $\kappa(n)$ and (ii) for every pair of nodes n and p in E, there is a path from n to p iff there is a path from $\kappa(n)$ to $\kappa(p)$ in E'. If there is an equivalence from E to E', then E and E' define the same synchronization problem.

3 Arbitration

We now explain what we mean by an arbiter and give some examples of synchronization that requires arbitration. We do not know any reasonable way to prove formally that a synchronization mechanism such as the wait/signal register does not require an arbiter. So, we will not attempt define to arbitration formally, and the definitions in this section are quite informal.

To simplify the exposition, we assume in this section a traditional totally-ordered (Newtonian) model of time in which all events are ordered by the time at which they occur. The time at which an operation execution begins or ends is the time of its earliest or latest event. Operation execution A precedes operation execution B iff A ends before B begins, and $A \dashrightarrow B$ implies that A begins before B ends.

3.1 Arbiters

An arbiter is a device that makes a discrete decision based on a continuous range of values. For our purposes, it suffices to consider the special case of an arbiter that makes a binary decision based on when some event occurs. So, we define an arbiter to be any device that produces an output value equal to 0 or 1, and for which there are two times T_0 and T_1 such that, if some particular event occurs at time T_i, then the arbiter's output value is i, for $i = 0, 1$. This is a looser definition of an arbiter than is customarily used by hardware designers.

Although we think of computer circuits as producing discrete outputs, they are actually continuous devices. A binary value is represented by the value of some physical quantity (usually a voltage) lying in one of two non-adjacent intervals. A simple argument shows that, if the arbiter's output is a continuous function of its input, then it can take an unbounded length of time to make its decision [9, 15]. This means that, for every Δ, there is a time T, lying between T_0 and T_1, and an $\epsilon > 0$ such that, if the event occurs between times $T - \epsilon$ and $T + \epsilon$, then the arbiter will not have decided by time $T + \Delta$.

The impossibility of building a bounded-time arbiter seems to be a fundamental law of physics, not a mathematical theorem. For example, Anderson and Gouda [1] proved that a bounded-time arbiter cannot be constructed from certain kinds of components, but their proof offers no insight into why the quantum-mechanical arbiter described in [9] doesn't work. We take the nonexistence of a bounded-time arbiter as an axiom.

As a corollary, we conclude that, if an arbiter is forced to decide within a bounded length of time, then it may produce an incorrect output—that is, a value other than 0 or 1. If the arbiter is a circuit, this could mean that it produces an output voltage that does not represent either a 0 or a 1. Another digital circuit receiving such a voltage as an input value could behave strangely. (One can design arbiter circuits for which the probability of not having decided within Δ seconds varies as $e^{-\Delta}$, so the probability of producing an incorrect output can be made vanishingly small by allowing enough time for the decision.)

A perfectly reliable arbiter can be implemented only with operations that may take an unbounded length of time to complete. We believe that any form of synchronization that cannot be implemented with bounded-time operations is equivalent to implementing an arbiter, but we know of no rigorous statement and proof of this. We will take arbitration-free synchronization to mean synchronization performed using bounded-time operations. The question we are addressing is, what synchronization problems can be solved by arbitration-free synchronization mechanisms?

3.2 A Safe Register Requires an Arbiter

A one-reader, one-writer one-bit safe register requires an arbiter. That is, it is impossible to implement it so that each operation takes a bounded length of time. To show this, we assume a one-reader, one-writer one-bit safe register in which all operations take at most δ time units, and we obtain a contradiction.

Suppose the writer performs just two writes: a write $W(0)^{[1]}$ of 0 that starts at time T_w and a write $W(1)^{[2]}$ of 1 that starts at time $T_w + 4\delta$. Suppose the reader performs a read operation R whose starting event occurs at time T. If $T = T_w + 2\delta$, then the assumption that each operation lasts at most δ time units implies that $W(0)^{[1]} \longrightarrow R \longrightarrow W(1)^{[2]}$, which by S1 (of Section 2.2) implies that R obtains the value 0. If $T = T_w + 5\delta$, then $W(1)^{[2]} \longrightarrow R$ and S1 implies that R obtains the value 1. By S2, the read must obtain either a 0 or a 1. Hence, the reader is an arbiter. Our assumption that an arbiter cannot always decide within a bounded length of time then contradicts the assumption that each read operation lasts at most δ time units.

3.3 Mutual Exclusion Requires an Arbiter

Although one can show directly that mutual exclusion requires arbitration, it is a little easier to use weak read/write registers mentioned in the introduction and described in detail in Section 7 below. A weak read/write register works correctly if there is mutually exclusive access to the register by the reading and writing processes. Thus, we can implement a safe read/write register that allows concurrent accesses by combining a weak read/write register with a mutual exclusion algorithm. A weak read/write register can be implemented without an arbiter, so an arbiter-free implementation of mutual exclusion would yield an arbiter-free implementation of a safe register. We have seen above that this is impossible, so we can conclude that there is no arbiter-free implementation of mutual exclusion.

3.4 Other Problems Requiring an Arbiter

There are a number of other classical synchronization problems that imply some form of mutual exclusion, including the readers-writers problem [4] and the dining philosophers problem [5]. They all require arbitration.

One synchronization problem that does not involve mutual exclusion but does require arbitration is concurrent garbage collection [6]. If an item becomes garbage before a collection phase begins, then it must be collected; if it becomes garbage after the collection phase ends, then it must not be collected. Since the item could become garbage at any time, deciding whether or not to collect it requires arbitration. However, there is also an important aspect of the problem that involves producer/consumer synchronization—namely, the collector consumes garbage and produces free items, while the mutator does the inverse. We show in Section 6 that producer/consumer synchronization does not require an arbiter. This implies that, although an arbiter is required to determine which items are garbage, concurrent access to the list of free items does not require arbitration.

3.5 Nondeterminism Requires an Arbiter

It seems that, even within a single process, implementing a truly nondeterministic choice (one that cannot in principle be predicted by executing a deterministic algorithm) requires an arbiter. However, we do not know how to prove this. Nondeterminism does not appear to be equivalent to arbitration, since allowing nondeterministic choice still does not permit a bounded-time arbiter. Nevertheless, all physical mechanisms we know of that make a nondeterministic choice require an arbiter, and hence cannot make the choice within a bounded length of time. For example, one way to choose a random bit is by determining if any nucleus in a piece of radioactive material decays within a fixed length of time—a procedure that is easily seen to require arbitration.

4 Wait/Signal Registers

A weak read/write register requires that different accesses to it not be concurrent. Preventing concurrent access by different processes requires synchronization. Hence, weak read/write registers cannot, by themselves, be used to synchronize processes. We now describe a general class of register, called a *wait/signal* register, that has an arbiter-free implementation and can be used to synchronize two independent processes. In such a register, a *receiver* process waits until a *sender* process has sent a signal.

We first define a use-once wait/signal register and characterize the synchronization problems that it can solve. We then define several types of reusable wait/signal registers that are all equivalent, in the sense that each can be used to implement the others. We also describe two arbiter-free implementations of wait/signal registers—one mechanical and another that uses the C-element defined in Section 2.3. The section concludes with an implementation of alternation synchronization with wait/signal registers. This implementation is the basic building block that we use in Section 6.1 to implement arbitrary marked-graph synchronization.

4.1 A One-Shot Wait/Signal Register

The simplest form of wait/signal register is one that can be used only once, which we call a *one-shot* wait/signal register. For such a register x, the sender can execute at most one $Signal(x)$ operation, and the receiver can execute at most one $Wait(x)$. The $Signal(x)$ operation execution always terminates in a bounded length of time. The $Wait(x)$ operation execution terminates only after a $Signal(x)$ operation execution has begun. More precisely, the $Wait(x)$ operation execution terminates iff there is a $Signal(x)$ operation execution, in which case $Signal(x) \dashrightarrow Wait(x)$ holds. Moreover, the $Wait(x)$ terminates within a bounded length of time after both it and the $Signal(x)$ have begun.

4.2 Implementing Systems with One-Shot Registers

A one-shot wait/signal register can be used to implement a single interprocess edge in an operation execution graph. (An interprocess edge is an edge that joins operation executions of different processes.) We implement the synchronization requirement implied by an edge from operation execution A of process p to operation execution B of process q as follows, using a one-shot wait/signal register x. Process p executes A followed by $Signal(x)$, and process q executes $Wait(x)$ followed by B. We then have

$$A \rightarrow Signal(x) \dashrightarrow Wait(x) \rightarrow B$$

where the $\rightarrow$ relations are guaranteed by the processes' algorithms and the $\dashrightarrow$ relation is guaranteed by the register x. Rule R2 then implies the relation $A \rightarrow B$ required by the edge.

With this simple procedure, we can implement any operation execution graph by using a separate one-shot wait/signal register for each interprocess edge. Conversely, we now show that a system in which deterministic processes communicate only with one-shot wait/signal registers implements a system described by an operation execution graph.

Proposition 1 *Let S be a multiprocess system consisting of deterministic processes communicating by one-shot wait/signal registers, and let $\widehat{S}$ be the* *system whose executions are obtained from those of S by removing Wait and Signal operation executions. Then $\widehat{S}$ consists of the set of system executions allowed by an operation execution graph.*

PROOF: Since each process is deterministic, it performs the same sequence of operation executions in any system execution. (A simple induction proof shows that a *Wait* terminates in one system execution iff it terminates in every system execution.) There can be only a single pair of $Wait(x)$ and $Signal(x)$ operation executions for each one-shot register x, and $Signal(x) \dashrightarrow Wait(x)$ if both operation executions occur. The possible executions of S are ones satisfying these $\dashrightarrow$ relations between corresponding *Signal* and *Wait* operation executions, together with the intra-process $\longrightarrow$ relations and all interprocess $\longrightarrow$ relations required by rules R1 and R2.

Let us call *Wait* and *Signal* operation executions the *synchronizing* operation executions. For each operation execution Op, let $Prec(Op)$ be the last non-synchronizing operation execution before Op by the same process, and let $Next(Op)$ be the next non-synchronizing operation execution after Op by the same process. (These operation executions need not exist.) All the $\longrightarrow$ relations implied by intra-process $\longrightarrow$ relations and the $\dashrightarrow$ relations are obtained by transitivity from the intra-process $\longrightarrow$ relations and all relations $Prec(Signal(x)) \longrightarrow Next(Wait(x))$ where the $Signal(x)$ and $Wait(x)$ operation executions both occur. We construct an operation execution graph whose nodes are the non-synchronizing operations and whose edges are determined by the intra-process $\longrightarrow$ relations and the relations $Prec(Signal(x)) \longrightarrow Next(Wait(x))$. The executions of $\widehat{S}$ are precisely the ones allowed by this operation execution graph. $\square$

If we restrict ourselves to a finite number of registers, then we can implement any finite operation execution graph. More generally, we can implement any operation execution graph that has only a finite number of interprocess edges. These are the only operation execution graphs that can be implemented using a finite number of one-shot wait/signal registers for interprocess communication. If we further restrict ourselves to finite-state processes, then the implementable synchronization problems are the ones described by an operation graph with a finite number of interprocess edges such that the sequence of operations executed by each process is either finite or repeating.

4.3 Resettable Wait/Signal Registers

Finite systems using one-shot wait/signal registers can implement synchronization problems with only a bounded number of operation executions, which are of little interest. To solve more interesting synchronization problems, we need a wait/signal register with which the sender and receiver can execute a (finite or infinite) sequence of matching *Signal* and *Wait* operations. A *resettable* wait/signal register permits this through the use of an additional *Reset* operation. The *Reset* operation must come between each successive pair of *Wait* and *Signal* operation executions. That is, suppose $W_x^{[i]}$ and $S_x^{[i]}$ are the i^{th} executions of the *Wait* and *Signal* operations, respectively, and $R_x^{[i]}$ is the i^{th} execution of the *Reset* operation. Then we require the following precedence relations:

$$
(6) \qquad
\begin{tikzcd}[row sep=small]
S_x^{[1]} \arrow[dr] & & S_x^{[2]} \arrow[dr] & & S_x^{[3]} \arrow[dr] & \\
& R_x^{[1]} \arrow[ur] \arrow[dr] & & R_x^{[2]} \arrow[ur] \arrow[dr] & & \cdots \\
W_x^{[1]} \arrow[ur] & & W_x^{[2]} \arrow[ur] & & W_x^{[3]} \arrow[ur] &
\end{tikzcd}
$$

These precedence relations must be guaranteed by the processes, in which case the register guarantees $S_x^{[i]} \dashrightarrow W_x^{[i]}$ for each i.

We define two types of resettable wait/signal registers, *sender-resettable* and *receiver-resettable*, depending on which process executes the *Reset*. For a sender-resettable register, (6) becomes:

$$
(7) \qquad
\begin{tikzcd}[row sep=small]
\text{Sender:} & S_x^{[1]} \arrow[r] & R_x^{[1]} \arrow[r] \arrow[dr] & S_x^{[2]} \arrow[r] & R_x^{[2]} \arrow[r] \arrow[dr] & S_x^{[3]} \arrow[r] & \cdots \\
\text{Receiver:} & W_x^{[1]} \arrow[rr] \arrow[ur] & & W_x^{[2]} \arrow[rr] \arrow[ur] & & W_x^{[3]} \arrow[ur] & \cdots
\end{tikzcd}
$$

For a receiver-resettable register, (6) becomes:

$$
(8) \qquad
\begin{tikzcd}[row sep=small]
\text{Sender:} & S_x^{[1]} \arrow[rr] \arrow[dr] & & S_x^{[2]} \arrow[rr] \arrow[dr] & & S_x^{[3]} \arrow[dr] & \cdots \\
\text{Receiver:} & W_x^{[1]} \arrow[r] & R_x^{[1]} \arrow[r] \arrow[ur] & W_x^{[2]} \arrow[r] & R_x^{[2]} \arrow[r] \arrow[ur] & W_x^{[3]} \arrow[r] & \cdots
\end{tikzcd}
$$

For both types of register, the horizontal $\rightarrow$ relations are achieved by having the processes execute the operations in the appropriate order; the diagonal $\rightarrow$ relations must be ensured by interprocess synchronization.

We now state the precise correctness condition SR for a sender-resettable wait/signal register. The definition is somewhat subtle. The precedence relations (7) are necessary for the register to be implementable. However, we can't simply require all these precedence relations to hold as a precondition for the register to work properly, because we must use these registers to implement the diagonal $\longrightarrow$ relations on which they depend. Instead, we need an inductive definition saying that if the operation executions are properly synchronized through the first $k-1$ *Wait* and *Signal* executions, then the k^{th} *Wait*, *Signal*, and *Reset* executions are correct. This condition is vacuous for $k=1$, so the first *Wait*, *Signal*, and *Reset* executions are guaranteed to work properly. For each $k>1$, we can use the correctness of the $(k-1)^{\text{st}}$ *Wait*, *Signal*, and *Reset* executions to prove the $\longrightarrow$ relations necessary for the correctness of the k^{th} executions. The correctness condition is:

SR. Assume that a sender and a receiver process execute the (finite or infinite) sequences of operation executions

$$\begin{array}{lccccccccc} \text{Sender:} & S_x^{[1]} & \longrightarrow & R_x^{[1]} & \longrightarrow & S_x^{[2]} & \longrightarrow & R_x^{[2]} & \longrightarrow & \cdots \\ \text{Receiver:} & W_x^{[1]} & & \longrightarrow & & W_x^{[2]} & & \longrightarrow & & \cdots \end{array}$$

For any $k \geq 1$, if $W_x^{[i]} \longrightarrow R_x^{[i]} \longrightarrow W_x^{[i+1]}$ for all i with $1 \leq i < k-1$, then:

1. If there is an $S_x^{[k]}$ operation execution, then it terminates.
2. If there is a $W_x^{[k]}$ operation execution, then it terminates iff there is an $S_x^{[k]}$ operation execution, in which case $S_x^{[k]} \dashrightarrow W_x^{[k]}$ holds.
3. If there is an $R_x^{[k]}$ operation execution, then it terminates.

The correctness condition RR for the receiver-resettable wait/signal register is analogous. It is obtained from SR by interchanging *sender* with *receiver* and W with R. We will not bother to write it out.

4.4 Implementing A Resettable Wait/Signal Register

We now show that a receiver-resettable wait/signal register can be implemented so the *Wait* and *Signal* operations take a bounded length of time. That is, condition RR holds with "terminating" strengthened to "completing within a bounded length of time". (For a *Wait*, this means within a bounded length of time of the beginning of the *Wait* or of the corresponding *Signal*, whichever occurs last.)

The signaling paradigm embodied in the wait/signal register is fundamental to the design of self-timed circuits [16], and it is well known that it can be implemented in silicon without an arbiter. An arbiter-free implementation with a C-element is described in Section 4.6 below. Here, we use a mechanical device to show that a bounded-time receiver-resettable

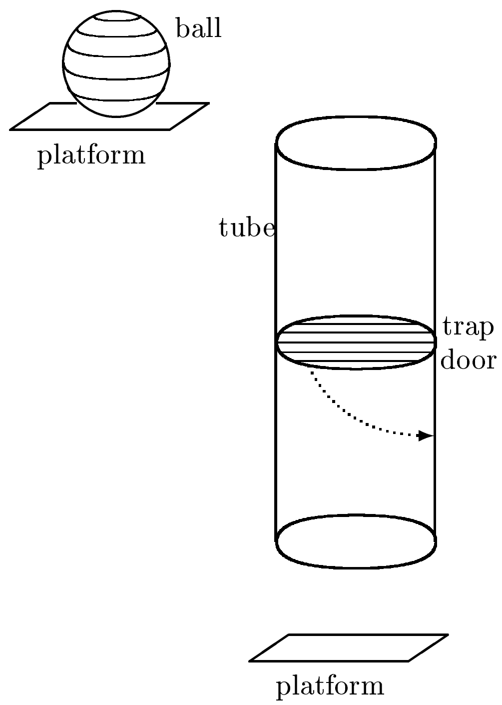


Figure 1: A mechanical device used to implement a wait/signal register.

wait/signal register can, in principle, be implemented. However, we do not analyze the design in enough detail actually to prove that it is a bounded-time device.

The heart of the device is shown in Figure 1. It consists of two platforms and a tube. The tube has a trap door that can be opened as indicated by the arrow. Initially there is a ball on the top platform and the trap door is shut. The *Signal* operation removes the ball from the top platform and drops it into the tube from the top. The *Wait* operation opens the trap door and waits for the ball to drop out the bottom of the tube and reach the bottom platform. The *Reset* operation moves the ball from the bottom platform to the top platform and shuts the trap door. The *Reset* operation is considered to be completed only after the trap door is closed and the ball is back on the top platform.

The assumption that the previous *Reset* is finished before a *Signal* is begun means that the *Signal* does not try to move the ball until after it has been placed on the platform. Hence, the *Signal* operation can easily be completed in a bounded length of time. Because the *Wait* operation starts only after the preceding *Reset*, it opens the trap door only when the ball is above the door—either on the platform, falling towards the door, resting on the door, or perhaps rising above the trap door after bouncing off the closed door. In any case, there is no resistance to opening the trap door, so that can be done in a bounded length of time. Once the trap door is open and the ball is in the tube, the ball will fall out the bottom of the tube in a bounded length of time. Moreover, it will hit the bottom platform with at least the velocity it would achieve by simply falling from the height of the trap door. Hence, the ball will have enough momentum to trigger a device that signals the receiver process that the *Wait* has completed and the *Reset* can begin. The entire *Wait* operation therefore completes in a bounded length of time after either it begins or the corresponding *Signal* begins. Because a *Reset* is not begun until the preceding *Wait* is completed, it does not try to close the trap door or move the ball until the ball is resting on the bottom platform. (We can assume that the bottom platform is made of a material that prevents the ball from bouncing.) Hence, the *Reset* can easily close the trap door and move the ball from the bottom to the top platform in a bounded length of time.

This implementation of a receiver-resettable wait/signal register without an arbiter depends on the assumption that the *Signal* and *Reset* operations are not concurrent. Otherwise, the two processes could concurrently be handling the ball on the top platform, and arbitration would be required to ensure that they didn't interfere with one another.

4.5 No-Reset Wait/Signal Registers

We can eliminate the *Reset* operations from a wait/signal register, as long as we keep the precedence relations between the *Signal* and *Wait* executions that they imply. This replaces (7) and (8) with

(9)
$$\begin{tikzcd} \text{Sender:} & S_x^{[1]} \arrow[r] \arrow[dr] & S_x^{[2]} \arrow[r] \arrow[dr] & S_x^{[3]} \arrow[r] \arrow[dr] & \cdots \\ \text{Receiver:} & W_x^{[1]} \arrow[r] \arrow[ur] & W_x^{[2]} \arrow[r] \arrow[ur] & W_x^{[3]} \arrow[r] \arrow[ur] & \cdots \end{tikzcd}$$

We define an NR wait/signal register to be one satisfying the following obvious analog of condition SR.

NR. Assume that a sender and a receiver process execute the sequences of operation executions $S_x^{[1]} \longrightarrow S_x^{[2]} \longrightarrow \cdots$ and $W_x^{[1]} \longrightarrow W_x^{[2]} \longrightarrow \cdots$, respectively. (The name indicates the direction of the diagonal arrows.) For any $k \geq 1$, if $W_x^{[i]} \longrightarrow S_x^{[i+1]}$ and $S_x^{[i]} \longrightarrow W_x^{[i+1]}$ hold for all i with $1 \leq i < k-1$, then

1. If there is an $S_x^{[k]}$ operation execution, then it terminates.
2. If there is a $W_x^{[k]}$ operation execution, then it terminates iff there is an $S_x^{[k]}$ operation execution, in which case $S_x^{[k]} \dashrightarrow W_x^{[k]}$ holds.

We can weaken the synchronization requirements for such a register by eliminating either the upwards or downwards pointing $\rightarrow$ relations from (9), getting:

(10)
$$\begin{tikzcd} \text{Sender:} & S_x^{[1]} \arrow[r] \arrow[dr] & S_x^{[2]} \arrow[r] \arrow[dr] & S_x^{[3]} \arrow[r] \arrow[dr] & \cdots \\ \text{Receiver:} & W_x^{[1]} \arrow[r] & W_x^{[2]} \arrow[r] & W_x^{[3]} \arrow[r] & \cdots \end{tikzcd}$$

(11)
$$\begin{tikzcd} \text{Sender:} & S_x^{[1]} \arrow[r] & S_x^{[2]} \arrow[r] & S_x^{[3]} \arrow[r] & \cdots \\ \text{Receiver:} & W_x^{[1]} \arrow[r] \arrow[ur] & W_x^{[2]} \arrow[r] \arrow[ur] & W_x^{[3]} \arrow[r] \arrow[ur] & \cdots \end{tikzcd}$$

Each of these possibilities yields a new type of register, which we call an *SW wait/signal register* and a *WS wait/signal register*, respectively. Their correctness properties SW and WS are obtained from NR by eliminating $W_x^{[i]} \rightarrow S_x^{[i+1]}$ (for SW) or $S_x^{[i]} \rightarrow W_x^{[i+1]}$ (for WS) from the hypothesis.

Conditions SW and WS each imply condition NR, since they have the same conclusion but weaker hypotheses. Hence SW and WS wait/signal registers are also NR wait/signal registers.

We have defined five classes of wait/signal register: sender-resettable, receiver-resettable, NR, SW, and WS. Section A.1 of the appendix shows that these classes are all equivalent in the sense that a register of any class can be implemented with registers of any other class.

4.6 Implementing a WS Wait/Signal Register

We now describe a simple implementation of a WS wait/signal register with a 2-input Muller C-element, described in Section 2.3. We let the sender be one input process, and we let the receiver be both the second input process and the C-element process. The sender's $S_x^{[i]}$ operation execution is implemented by the input process's $I_1^{[i]}$ operation execution. The receiver's $W_x^{[i]}$ operation execution is implemented by the sequence $I_2^{[i]} \rightarrow O^{[i]}$ of operation executions. The ordering assumptions $O^{[i]} \rightarrow I_j^{[i+1]}$ of the C-element are implied by the intraprocess order (for $j = 2$) and the assumption $W_x^{[i]} \rightarrow S_x^{[i+1]}$ of the wait/signal register (for $j = 1$). The requirements that

$W_x^{[i]}$ terminates only if $S_x^{[i]}$ occurs and that $S_x \dashrightarrow W_x^{[i]}$ follow from the C-element's guarantees that $O^{[i]}$ occurs only if $I_2^{[i]}$ does and that $I_2^{[i]} \dashrightarrow O^{[i]}$.

It is well-known in the self-timed circuit community that a C-element does not require an arbiter [19], so this is an arbiter-free implementation of the WS wait/signal register. The equivalence of all the different types of wait/signal registers, proved in the appendix, shows that any type of wait/signal register has an arbiter-free implementation.

4.7 Implementing Alternation

We now show how to implement alternation synchronization using wait/signal registers. We use two WS wait/signal registers, x and y, where the producer is the sender for x and the receiver for y, and the consumer sends with y and receives with x. The algorithm for the two processes is:

(12) Producer: **while** (true) $\{P;\ \mathit{Signal}(x);\ \mathit{Wait}(y)\}$

Consumer: **while** (true) $\{\mathit{Wait}(x);\ C;\ \mathit{Signal}(y);\}$

The proof of correctness of the algorithm is illustrated by the following diagram:

(13)
$$
\begin{tikzcd}
\cdots\ S_x^{[i-1]} \arrow[rr] \arrow[d, dashed] & & W_y^{[i-1]} \arrow[r] & P^{[i]} \arrow[r] & S_x^{[i]} \arrow[rr] \arrow[d, dashed] & & W_y^{[i]}\ \cdots \\
\cdots\ W_x^{[i-1]} \arrow[r] & C^{[i-1]} \arrow[r] & S_y^{[i-1]} \arrow[u, dashed] \arrow[rr] & & W_x^{[i]} \arrow[r] & C^{[i]} \arrow[r] & S_y^{[i]}\ \cdots \arrow[u, dashed]
\end{tikzcd}
$$

The horizontal solid arrows are implied by the algorithm. The vertical dashed arrows are obtained from the properties of the WS registers x and y. We first assume those dashed arrows and show that they imply that the P and C operation executions alternate as they are supposed to. We next prove that those dashed arrows are implied by condition WS.

To verify the safety property of alternation synchronization, we have to prove $P^{[k]} \rightarrow C^{[k]} \rightarrow P^{[k+1]}$ for all $k \geq 1$. The proof follows. (Refer to diagram (13) to help follow the proof.)

1. $P^{[k]} \rightarrow C^{[k]}$
   1.1. $P^{[k]} \rightarrow S_x^{[k]}$ and $W_x^{[k]} \rightarrow C^{[k]}$
   Proof: By the algorithm.
   1.2. $S_x^{[k]} \dashrightarrow W_x^{[k]}$
   Proof: Proved below.
   1.3. Q.E.D.
   Proof: Steps 1.1 and 1.2 and rule R2 of Section 2.1 imply $P^{[k]} \rightarrow C^{[k]}$.

2. $C^{[k]} \rightarrow P^{[k+1]}$

   2.1. $C^{[k]} \rightarrow S_y^{[k]}$ and $W_y^{[k]} \rightarrow P^{[k+1]}$

   PROOF: By the algorithm.

   2.2. $S_y^{[k]} \dashrightarrow W_y^{[k]}$

   PROOF: Proved below.

   2.3. Q.E.D.

   PROOF: Steps 2.1 and 2.2 and rule R2 imply $P^{[k]} \rightarrow C^{[k]}$.

To complete the proof of safety, we must prove 1.2 and 2.2. These properties follow from the conclusion of condition WS for registers x and y. To prove them, we must prove the hypothesis of WS, which states that the following two conditions hold for all i with $1 \leq i < k-1$:

(14) $W_x^{[i]} \rightarrow S_x^{[i+1]} \qquad W_y^{[i]} \rightarrow S_y^{[i+1]}$

The proof is by induction. We assume that (14) holds for $1 \leq i < k-2$ and prove it as follows for $1 \leq i = k-2$.

1. $W_x^{[k-2]} \rightarrow S_x^{[k-1]}$

   1.1. $W_x^{[k-2]} \rightarrow S_y^{[k-2]}$ and $W_y^{[k-2]} \rightarrow S_x^{[k-1]}$.

   PROOF: By the algorithm and the transitivity of $\rightarrow$ (rule R1).

   1.2. $S_y^{[k-2]} \dashrightarrow W_y^{[k-2]}$

   PROOF: This follows from condition WS for register y and the induction assumption, which asserts that (14) holds for $1 \leq i < k-2$.

   1.3. Q.E.D.

   PROOF: Steps 1.1 and 1.2 and rule R2 imply $W_x^{[k-1]} \rightarrow S_x^{[k]}$.

2. $W_y^{[k-2]} \rightarrow S_y^{[k-1]}$

   2.1. $W_y^{[k-2]} \rightarrow S_x^{[k-1]}$ and $W_x^{[k-1]} \rightarrow S_y^{[k-1]}$.

   PROOF: By the algorithm and the transitivity of $\rightarrow$ (rule R1).

   2.2. $S_x^{[k-1]} \dashrightarrow W_y^{[k-1]}$

   PROOF: The induction assumption asserts that $W_x^{[i]} \rightarrow S_x^{[i]}$ for $1 \leq i < k-2$. By step 1, this also holds for $i = k-2$, so it holds for $1 \leq i < k-1$. Condition WS for register x then implies $S_x^{[k-1]} \dashrightarrow W_y^{[k-1]}$.

   2.3. Q.E.D.

   PROOF: Steps 2.1 and 2.2 and rule R2 imply $W_y^{[k-2]} \rightarrow S_y^{[k-1]}$.

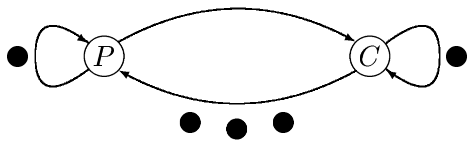
Thus far, we have ignored liveness. For alternation, there are two possible liveness properties: none or deadlock freedom, which is equivalent to process fairness. Whether or not our implementation satisfies deadlock freedom depends on the liveness assumption for the concurrent execution in the two-process algorithm (12). We obtain deadlock freedom if we assume deadlock freedom for the concurrent execution—namely, if no process is executing an operation and some process is ready to execute an operation, then some operation is eventually executed. To prove this, we must prove that each *Signal* and *Wait* execution that is begun eventually terminates. We have proved (14) for all i such that the operation executions in the formulas occur. A simple induction using condition WS for registers x and y then shows that all the *Signal* and *Wait* executions terminate, completing the proof.

5 Marked Graphs

Our main result is that the synchronization achievable by deterministic processes communicating with wait/signal registers is essentially described by marked graphs. This section provides the definitions and results about marked graphs needed to state and prove that result. It reviews the definition of a marked graph [3], defines its execution graph, and gives some properties of marked graphs and their execution graphs. We don't know if all these properties have already been published, but most of them would have been obvious to researchers working on marked graphs in the early 1970s. We also define process marked graphs to be a subclass of marked graphs that represent multiprocess systems.

5.1 Definitions and Simple Properties

A *marking* m of a directed graph G is a function that assigns a non-negative integer $m(\alpha)$ to each arc[3] α of G. We describe a marking m by saying that it places $m(\alpha)$ "tokens" on each arc α of G. We define the number of tokens on a path in a marking of G to the be sum of the number of tokens on each arc of the path. A *marked graph* is a pair $\langle G, m \rangle$ where G is a finite directed graph and m is a marking of G. Here is an example of a marked graph with two nodes, P and C.

(15) 

The path whose sequence of nodes is C, P, P, C, P contains 7 tokens.

For markings m and m' and node n of G, we write $m \xrightarrow{n} m'$ iff m' is obtained from m by removing one token from each input arc of n and

[3] To help avoid confusion, we use the term *arc* for an edge of a marked graph and reserve the term *edge* for execution graphs.

adding one token to each output arc of n. We usually describe $m \xrightarrow{n} m'$ as the assertion that firing n in marking m produces marking m'. A node n can be fired in marking m iff m puts at least one token on each input arc of n. In the marking (15), only node P can be fired. Firing it produces the marking obtained by removing one token from the bottom arc and putting one token on the top arc. (The token on the arc from P to itself is removed from the arc and then put back on it by firing P.)

A (finite or infinite) sequence $\langle n_1, n_2, \ldots \rangle$ of nodes is a *firing sequence* of a marked graph $\langle G, m \rangle$ iff there is a sequence of markings $\langle m, m_1, m_2, \ldots \rangle$ of G such that $m \xrightarrow{n_1} m_1 \xrightarrow{n_2} m_2 \ldots$. We call $\langle m, m_1, m_2, \ldots \rangle$ the sequence of markings of the firing sequence. A marked graph $\langle G, m \rangle$ is said to be *live* iff, for every node n of G, there is a firing sequence of $\langle G, m \rangle$ in which n appears infinitely often. The sequence $\langle P, P, C, P, C \rangle$ is a firing sequence of the marked graph (15). In general, a sequence of P's and C's is a firing sequence for this marked graph iff, in every prefix of the sequence, the number of P's minus the number of C's is between 0 and 3.

Perhaps the most important property of marked graphs is:

MG1. For any marked graph $\langle G, m \rangle$ and any cycle of G, the number of tokens on the cycle is the same in all the markings of any firing sequence of $\langle G, m \rangle$.

This property follows immediately from the observation that firing a node leaves the number of tokens on any cycle unchanged.

To help understand the possible firing sequences of a marked graph $\langle G, m \rangle$, we number the tokens and modify the firing rules as follows:

- The tokens on an arc form a queue, with the tokens initially on arc α numbered successively from $1 - m(\alpha)$ (at the head of the queue) through 0 (at the tail).

- The k^{th} firing of a node removes the token from the head of the queue on each of its input arcs and adds to the tail of the queue on each of its output arcs a token numbered k.

It is then easy to check the following property, where we let $Src(\alpha)$ and $Dest(\alpha)$ be the source and destination nodes, respectively, of an arc α.

MG2. For any arc α and $k \geq 1$, the k^{th} firing of $Dest(\alpha)$ removes token number $k - m(\alpha)$ from α.

5.2 Execution Graphs

The marked graph (15) describes the 3-buffer producer/consumer problem defined in Section 2.6. A sequence of P's and C's is a firing sequence for the marked graph iff the corresponding sequence of operation executions is a system execution (totally ordered by $\longrightarrow$) that is allowed by the 3-buffer problem. The operation execution graph (5) and the marked graph are related as follows: there is an edge from $n^{[i]}$ to $p^{[j]}$ in (5) iff there is an arc from node n to node p in the marked graph that contains $j - i$ tokens. For example, the edge from $C^{[2]}$ to $P^{[5]}$ in (5) corresponds to the arc from C to P in (15) containing three tokens.

We now generalize the construction of (5) from (15). For any marked graph $\langle G, m \rangle$, we define the *execution graph* $E(G, m)$ of $\langle G, m \rangle$ as follows:

- The nodes of $E(G, m)$ are all the elements of the form $n^{[i]}$ where n is a node of G and i is a positive integer.
- For each arc α of G and every $i \geq 1$, there is an edge from $\mathit{Src}(\alpha)^{[i]}$ to $\mathit{Dest}(\alpha)^{[i+m(\alpha)]}$.

Here are two properties relating a marked graph and its execution graph.

MG3. A marked graph $\langle G, m \rangle$ is live iff $E(G, m)$ is acyclic.

MG4. A sequence σ of nodes of G is a firing sequence of a marked graph $\langle G, m \rangle$ iff it satisfies the following condition for every arc α and all $k \geq 1 + m(\alpha)$: if there are k occurrences of $\mathit{Dest}(\alpha)$ in σ, then the $(k - m(\alpha))^{\text{th}}$ occurrence of $\mathit{Src}(\alpha)$ precedes the k^{th} occurrence of $\mathit{Dest}(\alpha)$ in σ.

Property MG4 follows from MG2. Property MG3 is proved by a simple induction argument using the following two properties, which are easy consequences of the definition of $E(G, m)$.

- Node n can be fired in $\langle G, m \rangle$ iff $n^{[1]}$ has no incoming edges in $E(G, m)$.
- $m \xrightarrow{n} m'$ implies that $E(G, m')$ is obtained from $E(G, m)$ by removing node $n^{[1]}$ (and its outgoing edges) and renaming node $n^{[i]}$ to $n^{[i-1]}$, for all $i > 1$.

5.3 Process Marked Graphs

Marked graphs describe synchronization among the firings of nodes. We can relate that to synchronization among operation executions by labeling each node with an operation, where firing a node corresponds to executing its operation. For a marked graph $\langle G, m \rangle$ to describe a multiprocess synchronization problem, the executions of operations belonging to an individual process must be totally ordered. This is guaranteed by the following condition:

PG. For every process p, the set of nodes labeled with an operation of p is the set of nodes of a cycle in G containing a single token.

We define a *process marked graph* to be a live marked graph $\langle G, m \rangle$ in which each node is labeled by an operation, and each operation is associated with one of a finite set of processes, such that PG holds. Requiring a process marked graph to be live allows us to prove the following result.

Proposition 2 *If $\langle G, m \rangle$ is a process marked graph, then $E(G, m)$ is an operation execution graph, where we associate with each node $n^{[i]}$ of $E(G, m)$ the operation labeling n.*

PROOF: To show that $E(G, m)$ is an operation execution graph, we must verify four conditions:

1. $E(G, m)$ is acyclic.
   PROOF: By MG3 and the assumption that $\langle G, m \rangle$ is live.
2. Each node in $E(G, m)$ has only a finite number of incoming and outgoing edges.
   PROOF: By the definition of $E(G, m)$.
3. Any infinite path has a first node.
   PROOF: By the definition of $E(G, m)$.
4. The nodes labeled with the operations performed by a process p form a path.
   PROOF: By PG and the definition of $E(G, m)$.

We next prove a necessary and sufficient condition for an operation execution graph to be generated by a process marked graph. But first, we need some definitions.

An *isomorphism* of operation execution graphs is defined in the obvious way to be a graph isomorphism that maps each node to a node with the same associated operation. An isomorphism is an equivalence, but the converse is not true in general. (Equivalence of operation execution graphs is defined in Section 2.6.) For ι to be an isomorphism, there must be an edge from n

to p iff there is one from $\iota(n)$ to $\iota(p)$; for it to be an equivalence, there must be a path from n to p iff there is one from $\iota(n)$ to $\iota(p)$.

For any function ι and any natural number k, we define ι^k to be the composition of ι with itself k times, where ι^0 is the identity function.

Let a *node subgraph* of a graph G be a subgraph consisting of some subset S of the nodes of G and all the edges in G whose source and destination are nodes in S. We define a *repeating isomorphism* on an operation execution graph E to be a graph isomorphism ι from E onto a node subgraph of E such that (i) there is no edge in E from a node in $\iota(E)$ to a node not in $\iota(E)$ and (ii) n and $\iota(n)$ have the same labels, for every node n of E.

Proposition 3 *If ι is a repeating isomorphism of an operation execution graph E, then there is a process marked graph $\langle G, m\rangle$ and an isomorphism κ from $E(G,m)$ to E such that $\iota^k(\kappa(n^{[i]})) = \kappa(n^{[i+k]})$ for every node n of G and all $i \geq 1$ and $k \geq 0$.*

PROOF: Let $\langle G, m\rangle$ be the operator marked graph such that (i) the set of nodes of G is the set of the nodes in E that are not in $\iota(E)$, (ii) for any nodes n and p of G and any $k \geq 0$, there is an arc from n to p containing k tokens iff there is an edge in E from n to $\iota^k(p)$, and (iii) the nodes of G are labeled with the operations of the corresponding nodes of E.

In an operation execution graph, the nodes associated with operations of each individual process form a path with a first node. It therefore follows from the definition of a repeating isomorphism that every node of E equals $\iota^k(n)$ for a unique node n in G and $k \geq 1$. We can therefore define a 1-1 correspondence κ from the nodes of $E(G,m)$ to the nodes of E such that $\kappa(\iota^k(n)) = n^{[k+1]}$ for all modes n of G and all $k \geq 0$. It is easy to check that κ is the required isomorphism. □

For any marked graph $\langle G, m\rangle$, the function ι defined by $\iota(n^{[k]}) = n^{[k+1]}$, for all nodes n of G and all $k \geq 1$, is a repeating isomorphism of $E(G,m)$. Proposition 3 therefore implies that an operation execution graph has a repeating isomorphism iff it is isomorphic to $E(G,m)$ for some process marked graph $\langle G, m\rangle$.

The following result will be used to reduce the problem of implementing an arbitrary process marked graph to that of implementing alternation. It asserts that, for any process marked graph, there is an equivalent one in which no arc ever has more than one token, and in which arcs come in oppositely-pointing pairs. This result is proved in Section A.2 of the appendix.

Proposition 4 *For any process marked graph $\langle G, m \rangle$, there is a process marked graph $\langle G', m' \rangle$ such that (i) for each arc α in G' there is an oppositely-pointing arc $\widehat{\alpha}$ such that m' assigns exactly one token to the cycle formed by α and $\widehat{\alpha}$, and (ii) $E(G', m')$ is equivalent to $E(G, m)$.*

5.4 Process Marked Graphs as Systems

We have described the firing of a node in a marked graph as an atomic action. However, we are interested in systems in which executing an operation is not necessarily an atomic action. We now show how to describe a process marked graph as a system, where firing a node consists of executing nonatomic operations that remove the tokens from the input arcs, perform the operation labeling the node, then add the tokens to the output arcs. Removing or adding a single token is a separate operation. The system executions of a process marked graph $\langle G, m \rangle$ are defined as follows.

Let O be the set of all operation executions of the form $n^{[k]}$, $RemT(\alpha)^{[k]}$, or $AddT(\alpha)^{[k]}$, where k is a positive integer, α is an arc of G and n is a node of G. ($RemT$ stands for *remove token* and $AddT$ for *add token.*) Let $n^{[k]}$ be an execution of the operation labeling n, let $RemT(\alpha)$ be an operation performed by the process associated with the destination node of α, and let $AddT(\alpha)$ be an operation performed by the process associated with the source node of α. We define a system execution of $\langle G, m \rangle$ to be any system execution whose operation executions are a subset of O such that the following constraints are satisfied, for every arc α and all $k \geq 1$:

- If there is a $Dest(\alpha)^{[k]}$ operation execution, then there is a $RemT(\alpha)^{[k]}$ operation execution and $RemT(\alpha)^{[k]} \longrightarrow Dest(\alpha)^{[k]}$ holds. (Tokens are removed from the input arcs before the node's operation is executed.)

- If there is an $AddT(\alpha)^{[k]}$ operation execution, then there is a $Src(\alpha)^{[k]}$ operation execution and $Src(\alpha)^{[k]} \longrightarrow AddT(\alpha)^{[k]}$ holds. (The node's operation is executed before tokens are added to the output arcs.)

- If there is a $RemT(\alpha)^{[k]}$ operation execution and $k - m(\alpha) \geq 1$, then there is an $AddT(\alpha)^{[k-m(\alpha)]}$ operation execution and $AddT(\alpha)^{[k-m(\alpha)]} \dashrightarrow RemT(\alpha)^{[k]}$ holds. (By MG2, this means the operation execution that removes a token must observe the operation execution that adds the token.)

Observe that applying R2 to the $\longrightarrow$ and $\dashrightarrow$ relations implied by these constraints yields a relation $n^{[i]} \longrightarrow p^{[j]}$ iff there is an edge from $n^{[i]}$ to $p^{[j]}$ in

$E(G, m)$. This observation leads to the following result, where the *synchronizing operations* are the *RemT* and *AddT* operations.

Proposition 5 *For any process marked graph $\langle G, m\rangle$, a set of executions of the operations labeling the nodes of G together with a $\longrightarrow$ relation satisfies the safety problem defined by the operation execution graph $E(G, m)$ iff it is the restriction to the non-synchronizing operation executions of a system execution of $\langle G, m\rangle$.*

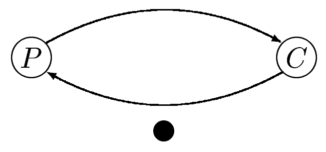
6 Implementation with Wait/Signal Registers

We now prove our main results, which describe the relation between wait/signal synchronization and process marked graphs. We first show that the synchronization problem defined by a process marked graph can be solved using wait/signal registers. We then show that a system composed of deterministic finite-state processes communicating with wait/signal registers implements the synchronization described by prefixing a finite operation execution graph to the execution graph of a process marked graph.

6.1 Implementing Marked-Graph Synchronization

By Proposition 3, an operation execution graph has a repeating isomorphism iff it is equivalent to the execution graph of a process marked graph $\langle G, m\rangle$. We now show how to solve the synchronization problem described by such an operation execution graph. By Proposition 5, we can solve the synchronization problem by implementing the corresponding marked graph system. By Proposition 4, we may assume that $\langle G, m\rangle$ is such that any arc α has an opposite-pointing arc $\widehat{\alpha}$, and the cycle formed by α and $\widehat{\alpha}$ has one token. (This additional constraint is not required by our construction, but it simplifies the proof.) We now solve the problem of implementing such a marked graph.

The alternation implementation (12) of Section 4.7 implements the following "two-cycle" marked graph:



Register x controls the token passing on the bottom arc and register y controls the token passing on the top arc. A *Wait* implements a *RemT* operation, and a *Signal* implements an *AddT* operation.

The marked graph that we must implement is a superposition of such two-cycles. We implement it by combining the implementations of all the two-cycles. We first rewrite the alternation implementation (12) as follows by unwinding the **while** loops a bit.

(16) Producer: $P;\ Signal(x);$
while $(true)\ \{Wait(y);\ P;\ Signal(x);\}$
Consumer: $Wait(x);\ C;\ Signal(y);$
while $(true)\ \{Wait(x);\ C;\ Signal(y);\}$

We now give an implementation of the synchronization constraints in which each node is a separate process. We use a separate wait/signal register for each arc. The program for a node n consists of a prefix followed by a **while** loop. The body of the **while** loop contains a $Wait(x)$ for each register x associated with an input arc of n, followed by the operation labeling node n, followed by a $Signal(y)$ for each register y associated with an output arc of n. The *Wait* operations may all be performed concurrently, as may the *Signal* operations. The prefix consists of the loop body, except with the *Wait* removed for each input arc containing a token in marking m.

If we ignore all operations other than the ones relevant to an individual two-cycle, we have the protocol (16), which we proved in Section 4.7 implements the required alternation synchronization. Hence, the implementation satisfies all the requirements of the original synchronization problem. However, we have implemented each node as a separate process. We now combine the algorithms of the individual nodes into an algorithm for each process.

By definition of a process marked graph, the nodes of a single process form a cycle containing a single token. Let $n_1, \ldots, n_k$ be the nodes of the cycle for a process p, where m places the token of the cycle on the input arc of n_1. In every firing sequence, the nodes of the process must fire in order, starting with n_1. So, we can combine the algorithms for these k nodes into a single algorithm with a prefix and a *while* loop, where the prefix and the loop body are the sequential compositions of the prefixes and loop bodies of all the nodes in order. Doing this for each process provides the desired implementation of the marked graph $\langle G, m \rangle$. (We can remove the operations for all registers corresponding to arcs that connect nodes of the same process; since the marked graph is live by definition, the synchronization implied by those arcs is subsumed by the sequential firing of the process's nodes.)

Our implementation of a process marked-graph $\langle G, m \rangle$ solves the safety part of the synchronization problem described by $E(G, m)$. To implement liveness properties, we have to assume fairness properties of our multiprocess algorithm. Define weak fairness for a set of processes to mean that, if some operation of one of the processes is enabled, then some operation from one of those processes must eventually be executed. We obtain a deadlock-free solution by assuming weak fairness for the entire set of processes. We obtain a process-fair solution by assuming weak fairness for each set of processes whose nodes all lie in the same connected component of G.

6.2 What is Implementable with Wait/Signal Registers

In any system, we can replace a reusable wait/signal register with an infinite sequence of one-shot wait/signal registers, replacing the k^{th} *Wait* or *Signal* operation with a *Wait* or *Signal* to the k^{th} one-shot register. We observed in Section 4.2 that the synchronization problems solvable with an arbitrary number of one-shot wait/signal registers are precisely those described by an operation execution graph. So, this is also the class that is solvable with an arbitrary number of reusable wait/signal registers. There remains the question of what class of operation execution graphs can be implemented with a finite number of reusable wait/signal registers.

In Section 4.2, we showed how to implement a finite operation execution graph with a finite number of wait/signal registers. (We can trivially substitute reusable registers for the one-shot registers used in that algorithm.) We just showed in Section 6.1 how to implement the execution graph of a process marked graph. We now show that, for finite-state processes, we can implement the class of synchronization problems obtained by combining a finite operation execution graph with a process marked graph.

Proposition 6 *A synchronization problem is solvable by a system of deterministic, finite-state processes using wait/signal registers iff it is described by an operation execution graph E with the following property: there is a finite set P of nodes of E such that (i) there is no edge from a node not in P to a node in P, and (ii) the node subgraph formed by the nodes not in P has a repeating isomorphism.*

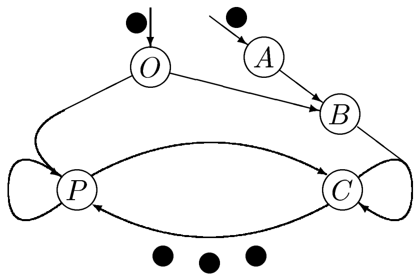
PROOF: We first sketch the proof of the "if" part. Let E_m be the node subgraph formed by the nodes not in P, and let ι be its repeating isomorphism. Choose a set N of nodes of E_m such that each node of E_m equals $\iota^k(n)$ for a unique n in N and a unique k. Choose r large enough so that there is no edge in E from a node in P to any node $\iota^k(n)$ for $k \geq r$.

Implement E_m with the algorithm described in Section 6.1 above. Next, unwind the first r iterations of each **while** loop, appending r copies of the body to the prefix code that precedes the loop. (If a process contains no nodes in E_m, that process is implemented with a *halt* operation, and there is no loop unrolling.) We then prepend to each process's algorithm the operations implementing the nodes of P and the *Wait* and *Signal* operations implementing the edges from nodes in P, as described in Section 4.2. (The *Wait* that implements an edge from a node in P to a node not in P is inserted in the appropriate place among the operations implementing E_m.) The resulting algorithm implements the synchronization problem described by E.

We now sketch the proof of the "only if" part. Let S be a system of deterministic, finite-state processes synchronizing only with reusable wait/signal registers. We use the same construction of an operation execution graph E as in the proof of Proposition 1, except using $\dashrightarrow$ relations only of the form $Signal(x)^{[k]} \dashrightarrow Wait(x)^{[k]}$ between the k^{th} executions of corresponding *Signal* and *Wait* operations. As in that proof, E describes all possible executions of the system $\widehat{S}$ whose operation executions consist of the executions of non-synchronization operations of S. Thus, E describes the synchronization problem solved by S. We must only show that E satisfies the required property.

Because S consists of a finite number of finite-state processes, the operation execution sequences of the processes, and their pattern of interaction (which *Wait* operation execution corresponds to which *Signal* operation execution) must eventually repeat. That is, by removing a finite prefix of E, we can obtain a node subgraph of E that has a repeating isomorphism. The omitted nodes form the required set P. $\square$

We can generalize process marked graphs to a class of automata that correspond precisely to the class of synchronization problems solvable by a system of deterministic, finite-state processes using wait/signal registers. We add to an ordinary process marked graph an acyclic graph of prefix nodes, where the prefix nodes of a process must fire before the non-prefix nodes. To connect the last non-prefix node of a process to the first non-prefix node of the process, we use a special two-source arc whose other source is the last non-prefix node of the process. (The order of a process's nodes in an ordinary marked graph is the order in which they can fire.) Firing either of its source nodes places a token on the arc. We also allow arcs with no source, so we can begin the execution of the prefix nodes of a process. Here is an example consisting of the two-process producer/consumer marked graph with an additional producer prefix node O and two additional consumer prefix nodes A and B.



Node O must fire before the first firing of node P; nodes A and B must fire before the first firing of node C. Node B cannot fire until nodes A and O have fired. Once the three prefix nodes have fired, this generalized marked graph acts like the ordinary marked graph (15).

In general, we require that the marked graph have a firing sequence in which each prefix node appears once and each non-prefix node appears infinitely often. We can then define the execution graph of a generalized process marked graph and restate Proposition 6 to assert that a synchronization problem is solvable by a system of deterministic, finite-state processes using wait/signal registers iff it is described by the execution graph of a generalized process marked graph. The interested reader can fill in the details.

7 Other Arbiter-Free Devices

We have characterized the synchronization achievable by deterministic processes communicating with wait/signal registers. This raises two questions: what extra power does nondeterminism add, and are there synchronization devices other than the wait/signal register that do not require arbitration?

Nondeterminism appears to be closely tied to the ability to communicate values. We do not know how to make any interesting use of nondeterministic choice unless one process can communicate the result of its choice to another process. Conversely, there is no need for a process to communicate a value unless that value arises from a nondeterministic choice. We introduce a weak read/write register, which allows processes to communicate values to one another. We start with a one-reader, one-writer one-bit register and use it to construct a multi-reader, multi-writer multi-valued register. We show that nondeterministic processes communicating with weak read/write registers and wait/signal registers can implement a simple generalization of process marked graphs. We conjecture that this is essentially all they can implement.

Finally, we introduce or-waiting, which is an arbitration-fee mechanism that also enlarges the class of synchronization problems that can be solved with wait/signal registers. We do not know any characterization of the class of synchronization problems that can be solved with them.

7.1 A Weak One-Reader, One-Writer One-Bit Register

We showed in Section 3.2 that a one-bit read/write register cannot be implemented with bounded-time operations. However, a computer depends on reading or writing one-bit registers (flip-flops) within a single clock cycle. There is no contradiction because those registers are not read and written concurrently. It is easy to implement registers if no two operations are concurrent.

We define a weak one-reader, one-writer one-bit register to be one satisfying condition S1 of the safe register defined in Section 2.2, except under the assumption that no read can be concurrent with a write. (This assumption makes S3 vacuous and, together with S1, it implies S2.) More precisely, the register allows the writer to perform a (finite or infinite) sequence $W(w_1)^{[1]} \longrightarrow W(w_2)^{[2]} \longrightarrow \cdots$ of terminating operation executions, where each w_i equals 0 or 1. For any j, if the reader performs a sequence of operation executions $R(r_1)^{[1]} \longrightarrow \cdots \longrightarrow R(r_j)^{[j]}$, none of which are concurrent with any of the $W(w_k)^{[k]}$, then $R(r_j)^{[j]}$ terminates and S1 holds.

A weak one-reader, one-writer one-bit register can be implemented with bounded-time operations. More precisely, the write operations can be executed in a bounded length of time, regardless of what the reader does. If none of the first j reads are concurrent with any writes, then each of those reads can also be executed in a bounded length of time.

7.2 General Weak Read/Write Registers

We can generalize the weak one-reader, one-writer one-bit register to a p-reader, q-writer n-bit register, for any p, q, and n. We assume that all the writes are totally ordered by $\longrightarrow$, so they form a (finite or infinite) sequence $W(w_1)^{[1]} \longrightarrow W(w_2)^{[2]} \longrightarrow \cdots$ where each w_i is a number from 0 to $2^n - 1$. Here, the $W(w_i)^{[i]}$ are all the writes by all the p writer processes. For any reader process and any j, if the process performs a sequence of operation executions $R(r_1)^{[1]} \longrightarrow \cdots \longrightarrow R(r_j)^{[j]}$, none of which are concurrent with any of the $W(w_k)^{[k]}$, then S1 holds.

We now show how to implement a p-reader, q-writer n-bit register with a collection of one-reader, one-writer one-bit registers. Unlike the comparable construction for general read/write registers, the construction for weak registers is quite simple and proceeds in three steps: constructing an n-bit register from one-bit registers, constructing a p-reader register from one-reader registers, and constructing a q-writer register from one-writer registers. These constructions can be applied in any order to implement a p-reader, q-writer n-bit register with a collection of one-reader, one-writer one-bit registers. The three constructions are:

- An n-bit register x is built from n one-bit registers $y[1], \ldots, y[n]$, where $y[i]$ holds the i^{th} bit of x.
- A p-reader register x is built from p one-reader registers $y[1], \ldots, y[p]$, where reader i reads $y[i]$, and a value v is written to x by writing it to each $y[i]$.
- A q-writer register x is built from q one-writer registers $y[1], \ldots, y[q]$, where $y[i]$ is written only by writer i. The value of x is $y[1] \oplus y[2] \oplus \cdots \oplus y[q]$, where $\oplus$ denotes bit-wise exclusive-or. The value of x is read by reading all the $y[i]$ and taking the exclusive-or. Writer i writes x by reading the values of $y[j]$ for all $j \neq i$ and then writing the appropriate value to $y[i]$.

If each writer is also a reader (so $p \leq q$), then this process implements a p-reader, q-writer n-bit register with $p * q * n$ one-reader, one-writer one-bit registers.

The correctness of these constructions follows easily from the definition (3) of $\longrightarrow$ for a high-level operation execution in terms of its component operation executions. This definition implies that, for each of the constructions, if $\mathcal{A}$ and $\mathcal{B}$ are executions of two operations to x such that $\mathcal{A} \longrightarrow \mathcal{B}$, and if A and B are executions of operations to $y[i]$ such that A is part of the implementation of $\mathcal{A}$ and B is part of the implementation of $\mathcal{B}$, then $A \longrightarrow B$. Hence, the assumption that the writes to x are totally ordered by $\longrightarrow$ implies that, for all i, the writes to $y[i]$ are also totally ordered by $\longrightarrow$, and if no read of x is concurrent with any write of x, then no read of $y[i]$ is concurrent with any write of $y[i]$.

7.3 Using Weak Read/Write Registers

Processes using a weak read/write register must be synchronized to prevent concurrent access to the register. Wait/signal registers can be used to implement this synchronization. Combining a weak read/write register with producer/consumer style synchronization, one process can transmit values to another. However, for deterministic processes, this accomplishes nothing because it is known in advance what values will be transmitted. To make use of weak read/write registers, we must add nondeterminism.

Implementing nondeterminism requires an arbiter. However, we can consider the nondeterministic choice to be made by the environment and given to the system as input. So, allowing a process to make a nondeterministic choice can be thought of as a way of modeling an arbiter-free system that accepts inputs from an environment that can use an arbiter. We therefore consider the synchronization problems that can be solved using wait/signal registers, weak read/write registers, and the operation of nondeterministically choosing one of a finite number of values. To describe these problems, we extend marked graphs to value-marked graphs as follows:

- We let each token have one of a finite number of values.[4]
- We label each arc with a set of operations and a "nondeterministic function" that, as a function of the values of the tokens removed from the input arcs, nondeterministically chooses an operation to perform and the values of the tokens placed on the output arcs.

A process value-marked graph is defined in the obvious way, with the requirement that all the operations labeling a node are operations of the same process.

The algorithm of Section 6.1 for implementing a process marked graph can be modified to implement a process value-marked graph. We again replace the graph by an equivalent one in which each arc can have at most one token. To each arc we associate a weak read/write register as well as a wait/signal register. The operation of putting a token with value v on the arc is implemented by writing v to the register and then performing the *Signal* operation. The operation of removing the token from the arc consists of first performing the *Wait* and then reading the value of the register. It is easy to show that this protocol guarantees that there are no concurrent operations to the register. The nondeterministic choice operation is used to decide what operation to execute and what values to put on the output tokens, as a function of the values read from the registers.

The generalization of process marked graphs described at the end of Section 6.2 can be applied to value-marked graphs as well. The implementation of an ordinary process value-marked graph can be extended to generalized

[4]The value of a token should not be confused with the number assigned to it, which is referred to in property MG2.

process value-marked graphs. We conjecture that generalized process value-marked graphs are the most general class of system that can be implemented by nondeterministic finite-state processes using wait/signal registers, weak read/write registers.

7.4 Or-Waiting

Waiting for a signal can be implemented without an arbiter. Waiting for either of two signals can also be implemented without an arbiter, as long as no attempt is made to determine which of the two signals occurred. If the *Wait* operation is implemented as a circuit that generates a signal when the operation completes, then combining the two outputs with an *or* gate implements waiting for either of the two signals. So, there is an arbiter-free implementation of the *or-waiting* operation $OrWait(x, y)$ whose execution completes if there is a matching execution of a $Signal(x)$ or $Signal(y)$ operation (or both). Completion of an execution OW of $OrWait(x, y)$ implies a relation $S \dashrightarrow OW$, where S is the corresponding execution of one of the two *Signal* operations.

When an $OrWait(x, y)$ operation completes, we know that either a $Signal(x)$ or a $Signal(y)$ operation has begun; but we cannot tell which. (To do so would require an arbiter.) If x and y are reusable registers, the *OrWait* must be followed by *Wait* operations to each register before the register can be reused. For every requirement $Op \longrightarrow W$ for the execution W of the following *Wait* operation, the relation $Op \longrightarrow OW$ must hold for the execution of the *OrWait*. For example, suppose x is a sender-resettable register. Using the notation of (7) in Section 4.3, if $OW^{[i]}$ is an execution of $OrWait(x, y)$ waiting for the execution $S_x^{[i]}$ of $Signal(x)$, then the algorithm must guarantee $R_x^{[i-1]} \longrightarrow OW^{[i]} \longrightarrow W_x^{[i]}$.

We generalize the *OrWait* operation in the obvious way to allow waiting for a *Signal* to any of n wait/signal registers. We allow multiple *OrWait* operations involving the same register to precede the *Wait* for that register. For example, a process could execute $OrWait(x, y)$, $OrWait(x, z)$, and $OrWait(y, z)$ before executing the next $Wait(x)$, $Wait(y)$, and $Wait(z)$ operations.

Let $Sig(x)$ be the predicate asserting that the next $Signal(x)$ has been issued, so $OrWait(x, y, z)$ waits for the predicate $Sig(x) \vee Sig(y) \vee Sig(z)$. Performing two *OrWaits* in sequence is equivalent to waiting for the conjunction of their conditions. For example, the sequence $OrWait(x, y)$; $OrWait(x, z)$ waits for $(Sig(x) \vee Sig(y)) \wedge (Sig(x) \vee Sig(z))$, which is equivalent to $Sig(x) \wedge (Sig(y) \vee Sig(z))$. Reduction to conjunctive normal allows us to implement with *OrWait* operations any expression formed by taking the conjunction and disjunction of *Sig* predicates.

Or-waiting extends the class of synchronization problems that can be solved with wait/signal registers. As an example, consider the following artificial (and useless) variant of alternation, in which there is one consumer and two producer processes performing the following sequence of operation executions:

$$\begin{array}{ll} \text{Producer 1:} & P_1^{[1]} \longrightarrow P_1^{[2]} \longrightarrow P_1^{[3]} \longrightarrow \cdots \\ \text{Producer 2:} & P_2^{[1]} \longrightarrow P_2^{[2]} \longrightarrow P_2^{[3]} \longrightarrow \cdots \\ \text{Consumer:} & C^{[1]} \longrightarrow C^{[2]} \longrightarrow C^{[3]} \longrightarrow \cdots \end{array}$$

The synchronization requirement is that, for each i, (a) $P_1^{[i]}$ or $P_2^{[i]}$ or both must precede $C^{[i]}$, and (b) $C^{[i]}$ must precede both $P_1^{[i+1]}$ and $P_2^{[i+1]}$.

This problem cannot be described by an operation execution graph, so it cannot be solved with only *Wait* and *Signal* operations. However, we can solve it using *OrWait*. We use two copies of the alternation algorithm (12), except we precede the consumer's C operation with an *OrWait* and move its *Wait* operations after the C operation:

$$\begin{array}{ll} \text{Producer 1:} & \textbf{while } (\mathit{true}) \ \{P;\ \mathit{Signal}(x1);\ \mathit{Wait}(y1)\} \\ \text{Producer 2:} & \textbf{while } (\mathit{true}) \ \{P;\ \mathit{Signal}(x2);\ \mathit{Wait}(y2)\} \\ \text{Consumer:} & \textbf{while } (\mathit{true})\{\mathit{OrWait}(x1, x2);\ C;\ \mathit{Wait}(x1);\ \mathit{Wait}(x2); \\ & \qquad\qquad \mathit{Signal}(y1);\ \mathit{Signal}(y2)\} \end{array}$$

The proof that this implements the synchronization requirement is similar to the proof that (12) implements alternation.

As this example shows, allowing or-waiting enlarges the class of synchronization problems that can be solved with wait/signal registers. We do not know how to characterize the class of problems solvable with or-waiting.

8 Conclusion

The impossibility of implementing arbitration in a bounded length of time seems to be a fundamental law of nature. Hence, what kind of synchronization can be achieved without arbitration should be a fundamental question in any theory of multiprocess synchronization. We know of no previous attempt to answer this question. Not coincidentally, we know of no practical benefits that might come from answering it. Nevertheless, we consider it to be an interesting question in its own right.

The problematic nature of arbiters has been recognized in the hardware community since at least the early 1970s [2]. There has been some consideration of arbiter-free circuits [21], but we know of no characterization of what can be implemented with such circuits. Moreover, the relation between arbiter-free circuits and multiprocess synchronization is unclear.

We believe that this article is the first to study arbiter-free multiprocess synchronization. We began by examining alternation, the simplest and most common form of arbiter-free synchronization, in which two processes take turns. For example, alternation is employed when one process transmits a sequence of values, one at a time, to another process. A typical implementation of alternation at the circuit level can be described in terms of wait/signal registers. With a wait/signal register, a receiver process can perform a *Wait* operation that waits for a sender process to perform a matching *Signal* operation. We identified five types of wait/signal registers, and showed that each can implement the others. Proposition 6 characterizes the class of synchronization implementable by deterministic, finite-state processes using wait/signal registers. It shows that this class is essentially the class of synchronization described by marked graphs [3].

Although a general read/write register requires an arbiter, no arbiter is needed to implement a weak read/write register that assumes a write is never concurrent with any other operation to the register. Adding weak read/write registers and nondeterministic choice to wait/signal registers allows the implementation of a generalization of marked graphs in which tokens have values and firing a node involves nondeterministic choice. We conjecture that such graphs essentially characterize the synchronization achievable by finite nondeterministic processes communicating with wait/signal registers and weak read/write registers.

We can also extend the capabilities of wait/signal registers, without adding arbitration, by allowing a process to wait for a signal to occur on any one of a set of registers. We do not know how to characterize the synchronization achievable with this extension.

Wait/signal registers, or-waiting, and weak read/write registers can all be implemented without arbiters. We do not know if there are other arbiter-free synchronization devices that cannot be implemented with them. Nor do we know how to characterize the multiprocess synchronization problems that can be implemented without an arbiter, independent of the synchronization mechanisms used. Much remains unknown in the theory of arbiter-free synchronization.

References

[1] James H. Anderson and Mohamed G. Gouda. A new explanation of the glitch phenomenon. *Acta Informatica*, 28(4):297–309, 1991.

[2] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Trans. Comput.*, C-22:421–422, April 1973.

[3] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(6):511–523, December 1971.

[4] P. J. Courtois, F. Heymans, and David L. Parnas. Concurrent control with “readers” and “writers”. *Communications of the ACM*, 14(10):667–668, October 1971.

[5] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[6] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[7] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[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. Buridan’s principle. Currently available from `http://research.microsoft.com/users/lamport/pubs/pubs.html`, or by searching the Web for the 23-letter string obtained by removing the - characters from all-lamports-pubs-onthe-web, January 1986.

[10] Leslie Lamport. The mutual exclusion problem—part i: A theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, January 1986.

[11] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.

[12] Leslie Lamport. On interprocess communication—part i: Basic formalism. *Distributed Computing*, 1:77–85, 1986.

[13] Leslie Lamport. On interprocess communication—part ii: Algorithms. *Distributed Computing*, 1:86–101, 1986.

[14] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1991.

[15] L. R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, C-30(2):107–115, February 1981.

[16] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, Massachusetts, 1980.

[17] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[18] Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer-Verlag, 1998.

[19] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[20] Tom Verhoeff. Analyzing specifications for delay-insensitive circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 172–183, San Diego, California, 1989.

[21] Tom Verhoeff. *A Theory of Delay-Insensitive Systems*. PhD thesis, Eindhoven University of Technology, May 1994.

Appendix

A.1 Equivalence of Wait/Signal Registers

A.1.1 Equivalence of Resettable Registers

We first demonstrate that sender-resettable and receiver-resettable registers are equivalent. We show how to implement a sender-resettable register x with two receiver-resettable registers $y1$ and $y2$. The converse construction of a receiver-resettable register from two sender-resettable registers is similar and is omitted.

The *Signal* executions of register x are implemented alternately by *Signal* executions of $y1$ and $y2$. The *Reset* executions of register x do nothing. Each

Wait of x is implemented by a *Reset* of one of the y registers and a *Wait* of the other. (The first two *Wait* executions of x are implemented just by *Wait* executions.) The construction and the proof are illustrated by the following diagram:

$$
(17)\quad
\begin{tikzcd}
\cdots & \overset{S_x^{[2i-1]}}{\boxed{S_{y1}^{[i]}}} \arrow[r] & \overset{R_x^{[2i-1]}}{\boxed{}} \arrow[r] \arrow[dr] & \overset{S_x^{[2i]}}{\boxed{S_{y2}^{[i]}}} \arrow[r] & \overset{R_x^{[2i]}}{\boxed{}} & \cdots \\
\cdots & \overset{W_x^{[2i-1]}}{\boxed{R_{y2}^{[i-1]},\ W_{y1}^{[i]}}} \arrow[ur] \arrow[rr] & & \overset{W_x^{[2i]}}{\boxed{R_{y1}^{[i]},\ W_{y2}^{[i]}}} \arrow[ur] \arrow[rr] & & \cdots
\end{tikzcd}
$$

The labeled boxes indicate how each operation on register x is implemented by operations on registers $y1$ and/or $y2$. For all i, operation execution $S_x^{[2i-1]}$ is implemented by operation execution $S_{y1}^{[i]}$, operation execution $R_x^{[2i-1]}$ is implemented by an operation that does nothing, $W_x^{[2i-1]}$ is implemented by the pair of operation executions $R_{y1}^{[i]}$ and $W_{y2}^{[i]}$, which may be concurrent, and so on. For $i = 1$ and $i = 2$, the $R_{y1}^{[0]}$ and $R_{y2}^{[0]}$ operations are omitted.

The arrows between boxes in diagram (17) are the relations on the operations of register x that we can assume. We have to use these assumptions and the correctness properties of registers $y1$ and $y2$ to prove the conclusions of SR, the correctness property of register x. We prove the conclusions of SR for $k = 2j$; the proof for k odd is similar. The hypothesis of SR asserts that the $\rightarrow$ relations in (17) hold for all the operation executions of x up to and including $S_x^{[2j]}$ and $W_x^{[2j]}$. We must prove the three conclusions of SR for $k = 2j$. These conditions, and their high-level proofs, are as follows. (The lower-level proofs of 1.1, 2.1, and 2.2 are given later.)

1. If there is an $S_x^{[2j]}$ operation execution, then it terminates.
   1.1. If $S_{y2}^{[j]}$ occurs, then it terminates.
   1.2. Q.E.D.
   Proof: Step 1.1 obviously implies the desired conclusion, since $S_x^{[2j]}$ is implemented by $S_{y2}^{[j]}$.
2. If there is a $W_x^{[2j]}$ operation execution, then it terminates iff there is an $S_x^{[2j]}$ operation execution, in which case $S_x^{[2j]} \dashrightarrow W_x^{[2j]}$ holds.
   2.1. If $R_{y1}^{[j]}$ occurs, then it terminates.
   2.2. If $W_{y2}^{[j]}$ occurs then it terminates iff $S_{y2}^{[j]}$ occurs, in which case $S_{y2}^{[j]} \dashrightarrow W_{y2}^{[j]}$ holds.

2.3. Q.E.D.

Steps 2.1 and 2.2 imply the desired conclusion because (i) an operation to x terminates iff all the operations that implement it terminate, and (ii) by (4) of Section 2.4, $S_{y2}^{[j]} \dashrightarrow W_{y2}^{[j]}$ implies $S_x^{[2j]} \dashrightarrow W_x^{[2j]}$.

3. If there is an $R_x^{[k]}$ operation execution, then it terminates.

PROOF: An operation that does nothing always terminates.

To complete the proof, we must prove 1.1, 2.1, and 2.2. These are the conclusions of condition RR for register $y2$, with j substituted for k. Since we are assuming that $y2$ satisfies RR, we just have to prove that the hypothesis of RR holds for $k = j$. That is, we must prove $S_{y2}^{[i]} \rightarrow R_{y2}^{[i]} \rightarrow S_{y2}^{[i+1]}$, for all i with $1 \leq i < j-1$. It's easy to read these $\rightarrow$ relations from the diagram above, since definition (3) of Section 2.4 asserts that a $\rightarrow$ relation between two boxes implies $\rightarrow$ relations among the operation executions within the boxes. Formally, we assume $1 \leq i < j-1$ and prove:

1. $S_{y2}^{[i]} \rightarrow R_{y2}^{[i]}$

1.1. $S_x^{[2i]} \rightarrow R_x^{[2i]} \rightarrow W_x^{[2i+1]}$

PROOF: This holds by hypothesis if $1 \leq 2i+1 \leq 2j$ (we are assuming the hypothesis of SR for $k = 2j$), and the assumption $1 \leq i < j-1$ implies $1 \leq 2i+1 \leq 2j$.

1.2. Q.E.D.

PROOF: The conclusion follows from step 1.1, the transitivity of $\rightarrow$, and (3), since $S_{y2}^{[i]}$ implements $S_x^{[2i]}$ and $R_{y2}^{[i]}$ is part of the implementation of $W_x^{[2i+1]}$ (because $i \geq 1$ implies $2i+1 > 2$).

2. $R_{y2}^{[i]} \rightarrow S_{y2}^{[i+1]}$

2.1. $W_x^{[2i+1]} \rightarrow R_x^{[2i+2]} \rightarrow S_x^{[2i+2]}$

PROOF: This holds by hypothesis if $1 \leq 2i+1$ and $2i+2 \leq 2j$, both of which follow from the assumption that $1 \leq i < j-1$.

2.2. Q.E.D.

PROOF: The conclusion follows from step 2.1, the transitivity of $\rightarrow$, and (3), since $R_{y2}^{[i]}$ is part of the implementation of $W_x^{[2i+1]}$ (because $i \geq 1$ implies $2i+1 > 2$) and $S_{y2}^{[i+1]}$ implements $S_x^{[2i+2]}$.

This completes the correctness proof for our implementation of a sender-resettable wait/signal register by two receiver-resettable ones.

A.1.2 Equivalence of No-Reset Registers

An SW or WS wait/signal register trivially implements an NR wait/signal register, since it is one. To show the equivalence of all three classes of register, we must show how to implement an SW and a WS register from NR registers. We implement a WS register; the implementation of an SW register is similar.

The implementation of a WS register x uses two NR registers y and z. It is described by the following diagram:

$$
\begin{array}{cccccc}
 & S_x^{[i-1]} & & S_x^{[i]} & & \\
\cdots & \boxed{S_y^{[i-1]} \rightarrow S_z^{[i-1]}} & \rightarrow & \boxed{S_y^{[i]} \rightarrow S_z^{[i]}} & \rightarrow \cdots & \\
 & W_x^{[i-1]} & \nearrow & W_x^{[i]} & \nearrow & \\
\cdots & \boxed{W_y^{[i-1]} \rightarrow W_z^{[i-1]}} & \rightarrow & \boxed{W_y^{[i]} \rightarrow W_z^{[i]}} & \rightarrow \cdots &
\end{array}
$$

An operation to x is implemented by the corresponding operation to y followed by the corresponding operation to z.

To prove the correctness of the implementation, we assume that y and z satisfy condition NR and that the hypothesis of WS holds for some value of k, and we prove that the two conclusions of WS hold for k. It is easy to check that the two conclusions of WS follow from the corresponding conclusions of NR for registers y and z. So, we just have to verify the hypothesis of NR for these registers. That is, we must prove $W_v^{[i]} \rightarrow S_v^{[i+1]}$ and $S_v^{[i]} \rightarrow W_v^{[i+1]}$ for v equal to y and to z, for all i with $1 \leq i < k-1$. The proof is by induction on k. We assume that these relations hold for all i with $1 \leq i < k-2$ and prove them for $1 \leq i = k-2$. The proofs are as follows.

1. $W_y^{[k-2]} \rightarrow S_y^{[k-1]}$

   PROOF: By the assumption that $W_x^{[k-2]} \rightarrow S_x^{[k-1]}$ (from the hypothesis of WS) and (3).

2. $S_y^{[k-2]} \rightarrow W_y^{[k-1]}$

   2.1. $S_y^{[k-2]} \rightarrow S_z^{[k-2]}$

   PROOF: By definition of the implementation of $S_x^{[i-1]}$.

   2.2. $S_z^{[k-2]} \dashrightarrow W_z^{[k-2]}$

   PROOF: By the induction assumption ($W_z^{[i]} \rightarrow S_z^{[i+1]}$ and $S_z^{[i]} \rightarrow W_z^{[i+1]}$ for $1 \leq i < k-2$) and condition NR for register z.

   2.3. $W_z^{[k-2]} \rightarrow W_y^{[k-1]}$

   PROOF: By (3) and the assumption $W_x^{[k-2]} \rightarrow W_x^{[k-1]}$.

   2.4. Q.E.D.

PROOF: Steps 2.1–2.3 and rule R2 of Section 2.1 imply $S_y^{[k-2]} \rightarrow W_y^{[k-1]}$.

3. $W_z^{[k-2]} \rightarrow S_z^{[k-1]}$

   PROOF: By (3) and the assumption $W_x^{[k-2]} \rightarrow S_x^{[k-1]}$.

4. $S_z^{[k-2]} \rightarrow W_z^{[k-1]}$

   4.1. $S_z^{[k-2]} \rightarrow S_y^{[k-1]}$

   PROOF: By (3) and the assumption $S_z^{[k-2]} \rightarrow S_z^{[k-1]}$.

   4.2. $S_y^{[k-1]} \dashrightarrow W_y^{[k-1]}$

   PROOF: By the induction assumption ($W_y^{[i]} \rightarrow S_y^{[i+1]}$ and $S_y^{[i]} \rightarrow W_y^{[i+1]}$ for $1 \leq i < k-2$) and condition NR for register y.

   4.3. $W_y^{[k-1]} \rightarrow W_z^{[k-1]}$

   PROOF: By definition of the implementation of $W_x^{[i-1]}$.

   4.4. Q.E.D.

   PROOF: Steps 4.1–4.3 and rule R2 imply $S_z^{[k-2]} \rightarrow W_z^{[k-1]}$.

A.1.3 Equivalence of Resettable and No-Reset Registers

To complete the proof of equivalence of our five types of register, it suffices to implement some type of no-reset register with a resettable register, and vice-versa. It is easy to implement a sender-resettable register with a WS register—just implement *Reset* with an operation that does nothing. It is easy to check that this implementation satisfies condition SR. We now show how to implement an NR register x using two SR registers y and z. The implementation and the hypotheses of NR used in the proof are described by the following diagram.

$$
(18) \quad
\begin{tikzcd}
\cdots & \overset{S_x^{[2i-1]}}{\boxed{R_{y2}^{[i-1]} \rightarrow S_{y1}^{[i]}}} \arrow[r] \arrow[dr] & \overset{S_x^{[2i]}}{\boxed{R_{y1}^{[i]} \rightarrow S_{y2}^{[i]}}} \arrow[r] \arrow[dr] & \cdots \\
\cdots & \overset{W_x^{[2i-1]}}{\boxed{W_{y1}^{[i]}}} \arrow[r] \arrow[ur] & \overset{W_x^{[2i]}}{\boxed{W_{y2}^{[i]}}} \arrow[r] \arrow[ur] & \cdots
\end{tikzcd}
$$

(For $i = 1$, the operation $R_{y2}^{[0]}$ is eliminated.)

To prove the correctness of the implementation, we assume that $y1$ and $y2$ satisfy condition SR and that the hypothesis of NR holds for some value of k, and we prove that the two conclusions of NR hold for k. We give the proof for $k = 2j$; the proof for k odd is similar. Here are the statements of those conclusions and their high-level proof outline.

1. If there is an $S_x^{[2j]}$ operation execution, then it terminates.
   1.1. If there are $R_{y1}^{[j]}$ and $S_{y2}^{[j]}$ operation executions, then they terminate.
   1.2. Q.E.D.
   PROOF: Step 1.1 implies the desired conclusion, since an operation on x terminates iff all its component operations do.
2. If there is a $W_x^{[2j]}$ operation execution, then it terminates iff there is an $S_x^{[2j]}$ operation execution, in which case $S_x^{[2j]} \dashrightarrow W_x^{[2j]}$ holds.
   2.1. If there is an $R_{y1}^{[j]}$ operation execution, then it terminates.
   2.2. If there is a $W_{y2}^{[j]}$ operation execution, then it terminates iff there is an $S_{y2}^{[j]}$ operation execution, in which case $S_{y2}^{[j]} \dashrightarrow W_{y2}^{[j]}$.
   2.3. Q.E.D.
   PROOF: The $W_x^{[2j]}$ operation execution occurs iff $W_{y2}^{[j]}$ occurs, and step 2.1 implies that $S_x^{[2j]}$ occurs iff $S_{y2}^{[j]}$ does. The desired conclusion then follows from step 2.2, since $S_{y2}^{[j]} \dashrightarrow W_{y2}^{[j]}$ implies $S_x^{[2j]} \dashrightarrow W_x^{[2j]}$ by (4).

To complete the proof, we must prove statements 1.1, 2.1, and 2.2. These all follow from the conclusions of condition SR for registers $y1$ and $y2$ and $k = j$. So, to prove them, we must prove the hypothesis of SR with $k = j$. Those hypotheses are that $W_v^{[i]} \rightarrow R_v^{[i]} \rightarrow W_v^{[i+1]}$ holds for v equal to $y1$ and to $y2$, and for $1 \leq i < j - 1$. It is easy to see from diagram (18) that these conditions are implied by (3) and the hypothesis of NR for register x (the inter-box arrows in the diagram above).

A.2 Proof of Proposition 4

We first choose $j \geq 1$ such that every arc of G is contained in a cycle that has at most j tokens. To see that such a j exists, observe first that if there were no cycle containing α, then $Dest(\alpha)$ could fire only a finite number of times, contradicting the assumption that $\langle G, m \rangle$ is a process marked graph, and hence is live. The existence of j then follows because a marked graph is finite by definition. Let $\oplus$ and $\ominus$ denote addition and subtraction modulo j.

By MG1, there can be at most j tokens on any arc throughout a firing sequence of $\langle G, m \rangle$. We construct $\langle G', m' \rangle$ by making j copies of each node and arc of G so that a marking of G with i tokens on an arc α corresponds to a marking of G' with one token on i of the copies of α. The precise construction is as follows. The nodes of G' consist of all nodes of the form

$\langle n, i \rangle$ where n is a node of G and $0 \leq i < j$. Each node $\langle n, i \rangle$ is labeled with the operation labeling n. There is an arc from $\langle n, i \rangle$ to $\langle n, i \oplus 1 \rangle$ for all n and i. Marking m' puts a token on this arc iff $i = j$. For every arc α of G and every i with $0 \leq i < j$, the graph G' also contains an arc $\langle \alpha, i \rangle$ from $\langle Dest(\alpha), i \ominus m(\alpha) \rangle$ to $\langle Src(\alpha), i \rangle$, for all i. Marking m' puts a token on this arc iff $0 \leq i < m(\alpha)$.

It can be shown that there is a 1-1 correspondence between the firing sequences of $\langle G', m' \rangle$, and $\langle G, m \rangle$ where:

- For any n and any i, the k^{th} firing of $\langle n, i \rangle$ corresponds to the $j * (k - 1) + i + 1^{\text{st}}$ firing of n, for any n and i, and
- For any α and i with $0 \leq i \leq j$, a marking of G having i tokens on arc α corresponds to a marking of G' such that, for some h, arc $\langle \alpha, h \oplus k \rangle$ has one token if $0 \leq k < i$ and no token if $i \leq k < j$.

We omit the proof.

It is easy to check that $\langle G', m' \rangle$ is a process marked graph. (The cycle in G containing the nodes of a process leads to a corresponding cycle j times as long in G'.) It then follows that the execution graphs $E(G', m')$ and $E(G, m)$ are isomorphic.

No arc ever contains more than one token during a firing sequence of $\langle G', m' \rangle$. Therefore, adding a backwards pointing arc $\widehat{\beta}$ for an arc β of G', with one token on the cycle formed by β and $\widehat{\beta}$, does not change the firing sequences of $\langle G', m' \rangle$. Adding all such arcs therefore leaves $E(G', m')$ the same up to isomorphism. This completes the proof of the proposition. $\square$