A Criterion for Atomicity

James H. Anderson*
Department of Computer Science
The University of Maryland at College Park
College Park, Maryland 20742-3255

Mohamed G. Gouda†
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

July 1990
Revised June 1991

Abstract

Most proof methods for reasoning about concurrent programs are based upon the *interleaving semantics* of concurrent computation: a concurrent program is executed in a stepwise fashion, with only one enabled action being executed at each step. Interleaving semantics, in effect, requires that a concurrent program be executed as a nondeterministic sequential program. This is clearly an abstraction of the way in which concurrent programs are actually executed. To ensure that this is a reasonable abstraction, interleaving semantics should only be used to reason about programs with "simple" actions; we call such programs "atomic." In this paper, we formally characterize the class of atomic programs. We adopt the criterion that a program is *atomic* if it can be implemented in a wait-free, serializable manner by a primitive program. A program is *primitive* if each of its actions has at most one occurrence of a shared bit, and each shared bit is read by at most one process and written by at most one process. It follows from our results that the traditionally accepted atomicity criterion, which allows each action to have at most one occurrence of a shared variable, can be relaxed, allowing programs to have more powerful actions. For example, according to our criterion, an action can read any finite number of shared variables, provided it writes no shared variable.

Keywords: atomicity, atomic registers, composite registers, history, interleaving semantics, leader election, models of concurrency, shared variable, snapshot, waiting

CR Categories: D.4.1, D.4.2, D.4.4, F.1.1, F.1.2

---

*Work supported, in part, at the University of Texas at Austin by Office of Naval Research Contract N00014-89-J-1913, and at the University of Maryland by an award from the University of Maryland General Research Board.

†Work supported, in part, by Office of Naval Research Contract N00014-89-J-1913.

1 Introduction

Most proof methods that have been proposed for reasoning about concurrent programs are based upon the assumption that only one action of a concurrent program is executed at a time [16, 21, 23, 26]. This model of program execution is often referred to as *interleaving semantics* since the actions executed by one process are interleaved with those of other processes. Under interleaving semantics, a concurrent program is executed as a nondeterministic sequential one. Therefore, techniques for reasoning about sequential programs (such as using invariants to prove safety properties and well-founded rankings to prove progress properties) can be applied to concurrent programs as well.

For interleaving semantics to be a reasonable abstraction of concurrent program execution, some restrictions must be imposed on the way in which concurrent programs are implemented on parallel machines. Clearly, it is unreasonable to require a concurrent program to be executed in a sequential fashion, i.e., one action at a time. Instead, such an implementation is typically expected to satisfy the weaker requirement that every parallel execution of two or more actions be "equivalent" to some interleaved one. For example, if a read and a write of a variable overlap in time, then the read either obtains the variable's value before the write or its value after the write — the net effect being that the read either precedes the write or vice versa. With programs implemented in this way, the power of parallel machines can be fully exploited, without having to abandon the simplicity of interleaving semantics.

The task of implementing a concurrent program so that any parallel execution is equivalent to an interleaved one may be difficult, unless we restrict our attention to programs with sufficiently simple actions. Consider, for example, a concurrent program consisting of processes $P_0, \ldots, P_{99}$ and variables $X_0, \ldots, X_{99}$; each process P_i, where $0 \leq i < 100$, consists of a single action $X_i := (\mathrm{SUM}\ j : 0 \leq j < 100 : X_j)$. If this program is to be executed in accordance with interleaving semantics, then a high degree of synchronization and coordination among processes will be required. We contend that such a program will not be "easy" to implement on any parallel machine.

To ensure that programs are easily implementable, interleaving semantics should not be used to reason about all programs, but only those programs with sufficiently simple actions; we call such programs "atomic." The purpose of this paper is to give a criterion for determining those programs that should be considered atomic.

If ease of implementation were our only concern, then we could stop here and define the class of atomic programs to include only those programs with the very simplest of actions. However, ease of implementation is not our only concern. We would also like to ease the burden on the programmer as much as possible by allowing a rich set of actions. Thus, we are faced with a dilemma: For ease of implementation, we should be as strict as possible, but for ease of programming, we should be as lax as possible. How can we possibly satisfy both of these seemingly conflicting requirements?

The roots of this question can be traced back to the seminal work of Owicki and Gries [26], who were perhaps the first to define a criterion for atomicity. In order to state their atomicity criterion, it is necessary to first distinguish between two classes of program variables, shared and private. A variable is *shared* if it appears among the actions of more than one process; otherwise, it is *private*. Owicki and Gries adopted the criterion that each action of a concurrent program has at most one occurrence of at most one shared variable. For example, if a is private and X and Y are shared,[1]

[1] We will generally follow the convention of using upper-case letters for shared variables, and lower-case letters for

then the action $X := a + 15$ would be allowed by their criterion, but the actions $X := Y + 2$ and $Y := Y + 1$ would not. The underlying assumption here is that it should be possible to read or write any shared variable in a single action. Owicki and Gries attribute the "observation that the memory reference must have 'reasonable' properties" to John Reynolds [26]. For this reason, the Owicki-Gries atomicity criterion is sometimes known as Reynolds' Rule [20]. This atomicity criterion has gained widespread acceptance since its introduction.

Owicki and Gries, in effect, resolve the dilemma noted above "by definition" because they do not defend their atomicity criterion on any formal basis. We are thus led to question their criterion on two grounds. First, it is not clear whether this is a viable criterion from an implementation standpoint. In fact, its "reasonableness" in this regard probably qualifies as a folk theorem [20]. Second, we question whether this criterion is unnecessarily restrictive. That is, it might be possible to relax the Owicki-Gries atomicity criterion somewhat without making programs any harder to implement.

In this paper, we remedy these two shortcomings by providing a formal framework for defining the class of atomic programs. This formal framework is based upon three concepts: primitive actions, wait-freedom, and serializability. The motivation behind this framework can best be explained by returning to the central question posed above: What kinds of actions should an atomicity criterion allow, given the conflicting requirements of ease of implementation and ease of programming?

Clearly, an atomicity criterion should allow one process of a concurrent program to communicate with another, i.e., it is unreasonable to disallow all actions that access shared variables. We therefore choose to allow some actions that access shared variables, but to ensure that programs are easily implementable, we will begin by insisting that each such action be as restrictive as possible. This gives rise to the notion of a "primitive" action: an action is *primitive* if it has at most one occurrence of a shared bit, and each shared bit is read by at most one process and written by at most one process. For example, letting a and b be private bits and X and Y be shared bits that are read by at most one process and written by at most one process, the action $a,\ b := X,\ \neg b$ would be considered primitive, but the actions $a,\ b := X, \neg X$ and $X := Y$ would not. Our starting point, then, is the assumption that each primitive action is reasonable enough to implement.

Given this assumption, it seems reasonable to allow certain nonprimitive actions as well. In particular, we also choose to allow any action that can be implemented in terms of primitive actions without a high degree of synchronization or coordination among processes. We illustrate this by considering two examples. First of all, consider the first nonprimitive action given above, i.e., $a,\ b := X,\ \neg X$. This action can be implemented in terms of primitive actions as follows.

$$a := X; \quad b := \neg a$$

Such an implementation seems sufficiently simple, so we conclude that our atomicity criterion should also allow an action such as $a,\ b := X,\ \neg X$.

Next, consider the other nonprimitive action mentioned above, i.e., $X := Y$. It turns out that this action can be quite difficult to implement in terms of primitive actions. In particular, consider the following concurrent program, where initially $X \neq Y$.

$$\textbf{process } P:\ X := Y \qquad\qquad \textbf{process } Q:\ Y := X$$

private variables.

According to interleaving semantics, either process P executes its action first or process Q does. In either case, the program terminates in a state satisfying the assertion $X = Y$. However, as shown in Section 5, if we implement each action $X := Y$ and $Y := X$ in terms of primitive actions, then at least one of the two processes P and Q must busy-wait in order to enforce the necessary mutual exclusion. A process that busy-waits may execute an infinite number of primitive actions, and hence may never terminate. Thus, it is impossible to implement the actions of this program in terms of primitive actions so that the program always terminates in a state satisfying $X = Y$. We therefore conclude that it is unreasonable to allow an action such as $X := Y$.

With this motivation in mind, our atomicity criterion can be described as follows. We define a program to be *atomic* if it can be implemented in a wait-free, serializable manner by a primitive program. A program is *primitive* iff each of its actions is primitive. Our notion of serializability is reflected by the implementation of the action $a,\ b := X,\ \neg X$ given above. In particular, this notion requires that whenever two or more implemented actions are executed concurrently (i.e., the primitive actions that implement them are interleaved), there exists an equivalent sequential execution.

Any program allowed by our atomicity criterion is easily implementable because it can be "reduced" to one with only primitive actions. It turns out that our atomicity criterion includes all programs allowed by the Owicki-Gries atomicity criterion.[2] This confirms their choice of actions from an implementation standpoint. However, as we shall see shortly, our atomicity criterion allows some rather complicated actions that are not allowed by the Owicki-Gries criterion. For example, our criterion allows an action to read any finite number of shared variables, provided it writes no shared variable. Thus, our results establish that the traditional Owicki-Gries atomicity criterion is too restrictive.

The rest of this paper is organized as follows. In Section 2, we present our model of concurrent programs, and in Section 3, we present the proposed atomicity criterion mentioned above. The task of checking whether a program is atomic is nontrivial if we appeal directly to this criterion. Therefore, we present a method that allows this task to be performed systematically. We outline our approach to justifying this method at the end of Section 3. The actual justification is given in Sections 4 through 8. Concluding remarks appear in Section 9.

2 Concurrent Programs

A *concurrent program* consists of a set of processes and a set of variables. A *process* is a sequential program specified using Dijkstra's guarded commands [12]. Each *variable* is either unsubscripted or subscripted. Unsubscripted variables are defined using integer, boolean, and subrange (e.g., $0..K$) types. Subscripted variables are defined using an array type; the elements of an array are of type integer, boolean, or subrange. Multi-dimensional arrays and arrays with an infinite number of elements are allowed.

Each variable of a concurrent program is either private or shared. A *private variable* is defined only within the scope of a single process, whereas a *shared variable* is defined globally and may be

[2] At least, for the kinds of programs considered here. We require processes to be specified using ordinary sequential programming statements, whereas Owicki and Gries also allow the possibility of an **await** statement. Such a statement obviously cannot be implemented from primitive actions in a wait-free fashion, and hence should not be allowed.

accessed by more than one process. We assume that the elements of an array are either all shared or all private to the same process. We also make the simplifying assumption that only expressions over private variables are used as array subscripts.

In the remainder of this paper, the term “variable” is meant to refer to either an unsubscripted variable or an element of an array — we do not refer to an entire array as being a single variable.

The processes of a concurrent program are allowed to access shared variables only by executing assignment statements. An assignment statement has the form

$$x_1,\ \ldots,\ x_N := E_1,\ \ldots,\ E_N$$

where each x_i is a distinct variable and each E_j is an expression over a finite number of variables. This statement is executed as a single action as follows: first, the expressions $E_1,\ \ldots,\ E_N$ are evaluated along with any array subscripts appearing in $x_1,\ \ldots,\ x_N$; then, the value of each expression E_i is assigned to the corresponding variable x_i. We say that the variables appearing in the expressions $E_1,\ \ldots,\ E_N$ and the array subscripts appearing in $x_1,\ \ldots,\ x_N$ are *read* by the assignment statement, and the variables $x_1,\ \ldots,\ x_N$ are *written* by the assignment statement. For example, consider the assignment statement $X[i],\ i := Y[j],\ j+1$. This statement reads the variables i, j, and $Y[j]$, and writes the variables i and $X[i]$.

Associated with each shared variable is a set of processes that are allowed to *read* that variable, and a set of processes that are allowed to *write* that variable. These two sets of processes must be specified as part of the definition of every shared variable. An assignment statement of a process may read (write) a shared variable only if that process has read (write) access to that variable. (Presumably, a runtime error would occur if this rule is ever violated.)

3 Atomicity Criterion

In this section, we define two important classes of programs, “primitive” and “atomic.” Informally, a program is primitive if its processes communicate only by means of single-reader, single-writer, shared bits. More precisely, a program is *primitive* iff it satisfies the following six constraints.

(0) Each shared variable is a single bit.

(1) Each shared variable is read by at most one process.

(2) Each assignment statement reads at most one shared variable and does so at most once.

(3) Each shared variable is written by at most one process.

(4) Each assignment statement writes at most one shared variable.

(5) An assignment statement does not both read a shared variable and write a shared variable.

Note that these constraints can be checked syntactically. In particular, constraint (0) can be checked by simply noting the type of each shared variable. Constraints (1) and (3) can be checked by counting the number of processes that are allowed to read or write each shared variable. Constraints (2), (4), and (5) can be checked by counting the number of occurrences of shared variables in each assignment statement. This can be done on a syntactic basis, given our assumptions concerning variable declarations. (For example, consider the statement $a,\ b := X[i],\ X[j]$, where a and b are private. By assumption, either all elements of the array X are shared, or all are private to the same process. If each element of X is shared, then there are two occurrences of shared variables in this statement, namely $X[i]$ and $X[j]$; if, on the other hand, each element of X is private, then there are no occurrences of shared variables in this statement.)

As stated previously, we assume that any primitive program is easily implementable. Our definition of the more general class of atomic programs reflects this assumption. Simply stated, this definition is as follows.

> *Atomicity Criterion*: A program is *atomic* iff it can be implemented by a primitive program.

The notion of "implements" referred to here is formally defined in Section 4. Informally, we implement an atomic program by replacing each assignment statement that fails to satisfy constraints (0) through (5) by a program fragment whose statements satisfy (0) through (5). Different program fragments in different processes may be executed concurrently (i.e., their statements may be interleaved); each such execution, however, is required to "resemble" one in which the program fragments are executed in sequence. The program fragments are restricted to be wait-free, i.e., busy-waiting loops that can iterate an infinite number of times are not allowed. This restriction guarantees that a process executes each of its implemented assignment statements in a finite amount of time, regardless of the activities of other processes.

In general, the task of proving that a given program satisfies our proposed atomicity criterion is nontrivial. A proof must establish that whenever two or more of the above-mentioned program fragments in different processes are executed concurrently, there exists an equivalent execution in which they are executed in sequence. The number of different cases that must be taken into account in order to establish this proof obligation can be considerable (see, for example, the proofs given in [4]).

The main contribution of this paper is to give a simple method for checking whether a given program satisfies the proposed atomicity criterion. In many cases, this method amounts to a syntactic check of the program text. (For instance, see the example below.) In particular, we prove that a program is atomic if its assignment statements satisfy the following three conditions.

- *Exact-Writing*: The shared variables can be partitioned into classes, where each class either consists of a single variable, or a number of unsubscripted variables, such that the following condition holds: if an assignment statement writes a variable from some class, then it writes every variable in that class and no variable from any other class.

- *Single-Phase*: If an assignment statement of one process reads a variable that is written by another process, then that assignment statement does not write a shared variable.

- *Finite-Reading*: If an assignment statement reads an element of an infinite shared array, then it reads no other shared variable.

We call these three conditions *atomicity constraints*. The Exact-Writing and Finite-Reading constraints can be checked on a syntactic basis by simply checking the shared variables that are written and read, respectively, by each assignment statement. By contrast, it may be necessary to take the semantics of a program into account in order to check whether Single-Phase is satisfied. For example, consider the assignment statement $X[i] := Y[j]$ of some process P. To determine whether this statement violates Single-Phase, it is necessary to determine whether $Y[j]$ is written by a process other than P — but, this may depend on the value of j. Note that we are, in effect, viewing each statement such as $X[i] := Y[j]$ as being a separate statement for each of the possible values of i and j.

The Exact-Writing constraint specifies the conditions under which an assignment statement may write several shared variables; an assignment statement that writes several shared variables is called a *multi-register assignment* in [14]. Intuitively, this constraint implies that we can view the set of shared variables written by a given assignment statement as a single "compound" variable. (For the sake of simplicity, we have made Exact-Writing more restrictive than is necessary. In particular, we allow a class to consist of more than one shared variable only if each variable in the class is unsubscripted. In [4], a more permissive version of Exact-Writing is considered in which a class may consist of several subscripted variables. This version of Exact-Writing is considered in Section 9.) The Single-Phase constraint specifies the conditions under which an assignment statement may both read and write shared variables. As shown in Section 6, this constraint implies that any assignment statement can easily be implemented in terms of assignment statements that either read shared variables or write shared variables; hence the name "Single-Phase." The Finite-Reading constraint specifies the conditions under which an assignment statement may read several shared variables. Note that this is a very permissive constraint, as it only restricts those programs that have infinite shared arrays.

We illustrate the atomicity constraints by the following example.

Example: Consider the concurrent program in Figure 1. This program consists of a *CONTROLLER* process and N *DEVICE* processes that communicate by means of five shared variables V, W, X, Y, and Z. Variables V, X, and Y are written by the *CONTROLLER*, and variables W and Z are written by the *DEVICE*s. Variables V and X are read by the *DEVICE*s, variable W is read by the *CONTROLLER*, and variables Y and Z are read by all $N + 1$ processes.

The five variables V, W, X, Y, and Z are used as follows. V stores a "request" as produced by the *CONTROLLER* and X identifies the *DEVICE* for which the request is intended. The particular *DEVICE*'s "reply" to the given request is stored in W. A request is "pending" iff $Y \neq Z$; Y is updated by the *CONTROLLER* and Z by the *DEVICE*s. The protocol followed by these processes is simple. The *CONTROLLER* repeatedly consumes replies and produces requests, and each *DEVICE* repeatedly consumes requests and produces replies.

To verify that Exact-Writing is satisfied, we partition the shared variables into classes as follows: the tuple $\{V, X, Y\}$ is a class, and the pair $\{W, Z\}$ is a class. Observe that the only assignment statements that write shared variables are

$$V,\ X,\ Y := request,\ dev,\ \neg Y$$

and

$$W,\ Z, := reply,\ z\quad .$$

```
shared var V, W : integer;
           X : 1..N;
           Y, Z : boolean

initialization (W = default value) ∧ (Y = Z)

process CONTROLLER
private var request, reply : integer;
            dev : 1..N;
            pending : boolean
begin
    do true  → reply, pending := W, (Y ≠ Z);
               if ¬pending  → consume reply;   produce next request, dev;
                              V, X, Y := request, dev, ¬Y
               [] pending  → skip
               fi
    od
end

process DEVICE(j : 1..N)
private var request, reply : integer;
            dev : 1..N;
            pending, z : boolean
begin
    do true  → request, dev, pending := V, X, (Y ≠ Z);
               if pending ∧ dev = j  → consume request;   produce reply;
                                       z := ¬Z;
                                       W, Z := reply, z
               [] ¬pending ∨ dev ≠ j  → skip
               fi
    od
end
```

Figure 1: An example program.

Thus, because V, X, and Y are always written together, and because W and Z are always written together, Exact-Writing is satisfied. Note that the only assignment statement that both reads and writes shared variables is the statement

$$V,\ X,\ Y := request,\ dev,\ \neg Y$$

of the $CONTROLLER$. Because variable Y is written only by the $CONTROLLER$, this implies that Single-Phase is satisfied. Finite-Reading is trivially satisfied because there are no infinite shared arrays. Notice that this program violates the traditional atomicity criterion of Owicki and Gries. □

The remaining sections are dedicated to justifying the proposed atomicity constraints: Exact-Writing, Single-Phase, and Finite-Reading. In particular, we show that any program satisfying these atomicity constraints is atomic. We also show that Exact-Writing and Single-Phase are necessary in the following sense: for each of these constraints, there exists a nonatomic program violating that constraint, but satisfying the other two constraints. This shows that if we remove either of these two constraints from our set of constraints, then we can no longer make the claim that every program satisfying our set of constraints is atomic. As yet, we do not know whether Finite-Reading is necessary in this sense. However, this constraint does not appear to be a severe limitation, as it only restricts those programs that have infinite shared arrays.

To establish the necessity of Exact-Writing and Single-Phase, we consider in Section 5 the problem of two-process leader election. We first show that there is no primitive program that solves this problem. Then, we exhibit two programs that solve this problem, one that violates Exact-Writing and satisfies Single-Phase and Finite-Reading, and another that violates Single-Phase and satisfies Exact-Writing and Finite-Reading. Neither of these two programs can be implemented by a primitive program, because we know that such a program does not exist. Thus, both programs are nonatomic. This establishes the following theorem.

Theorem 1: There exists a nonatomic program violating Exact-Writing (or Single-Phase), but satisfying the other two constraints. □

To establish the sufficiency of our atomicity constraints, we consider in Section 6 a shared data object called a *composite register* [2, 3]. A composite register is an array-like variable consisting of a number of components. An operation of the register either writes a value to a single component, or reads the values of all of the components. As shown in Section 6, any program satisfying our proposed atomicity constraints can be implemented by a program with processes that communicate only by reading and writing composite registers. Thus, we can establish the sufficiency of our atomicity constraints by showing that a composite register can be implemented in terms of single-reader, single-writer, shared bits. Given the structure of a composite register, this proof obligation can be broken into several steps. The register constructions for these steps are described in Section 6; one of the constructions is presented in detail in Section 8. These constructions establish the following theorem.

Theorem 2: Any program satisfying Exact-Writing, Single-Phase, and Finite-Reading is atomic. □

4 Implementing Atomic Programs

As stated in Section 3, a program is atomic iff it can be implemented by a primitive program. In this section, we formally define what it means to implement one program by another.

Terminology: If program A is implemented by program B, then we refer to A as the *implemented program*, and B as the *implementation*. □

Definition: An *implementation* is obtained by replacing each shared variable of the implemented program by a set of variables, and each assignment statement of the implemented program by a program fragment. In principle, such a program fragment can be expressed as follows:

$$S_0;\ \mathbf{do}\ B_1\ \rightarrow\ S_1\ [\!]\ \cdots\ [\!]\ B_N\ \rightarrow\ S_N\ \mathbf{od}$$

where each B_i is an expression over private variables, and each S_j is an assignment statement. $\Box$

As we shall see later, an implementation is *correct* iff it is wait-free, sequentially correct, and serializable. Informally, these conditions are defined as follows.

- *Wait-Freedom*: Each program fragment is free of busy-waiting loops that can potentially iterate an infinite number of times.
- *Sequential Correctness*: When executed in isolation, each program fragment has the same "effect" as the assignment statement that it replaces.
- *Serializability*: Whenever two or more program fragments in different processes are executed concurrently (i.e., their statements are interleaved), there exists an "equivalent" execution in which the program fragments are executed in sequence.

In this section, we formally define these conditions. We motivate our formal treatment by considering the following example. As this example shows, it is usually straightforward to define an implementation so that wait-freedom and sequential correctness hold. The principal difficulty lies in satisfying serializability.

Example: Consider a concurrent program with a process that includes the assignment statement $X := 4$, where X is a single-reader, single-writer, four-bit, shared variable. Note that this statement violates constraint (0) in the definition of primitive programs given in Section 3. To implement the given program by a primitive program, it is necessary to replace X by a set of single-bit variables. One way to do this is to partition variable X into four separate single-bit variables X_0, X_1, X_2, and X_3. Then, we can replace the statement $X := 4$ by the following program fragment.

$X_0 := 0;$
$X_1 := 0;$
$X_2 := 1;$
$X_3 := 0$

Each other assignment statement of the implemented program that reads (or writes) X would similarly be replaced by a program fragment that reads (or writes) each X_i.

The above program fragment for $X := 4$ is loop-free. Moreover, executing this program fragment in isolation clearly has the effect of assigning the value 4 to variable X. Thus, this implementation is both wait-free and sequentially correct. However, a read of X may obtain an incorrect value if it is executed concurrently with a write to X. Suppose, for example, that the program fragment for $X := 4$ is executed when $X = 15$ (i.e., when $X_0 = X_1 = X_2 = X_3 = 1$). If a program fragment that reads X (by reading each X_i) is executed between the write of 0 to X_1 and the write of 1 to X_2, then the incorrect value 12 will be obtained for X (i.e., $X_0 = X_1 = 0$ and $X_2 = X_3 = 1$). Thus, this simple implementation is not serializable. □

Terminology: In order to avoid confusion, we henceforth refer to the variables of an implemented program as *higher-level* variables, and the variables of an implementation as *lower-level* variables. Note that each higher-level variable corresponds to some set of lower-level variables. We capitalize terms such as "Read" and "Write" when they apply to higher-level variables, and leave them uncapitalized when they apply to lower-level variables. □

In the above example, the higher-level variable X corresponds to the set of lower-level variables $\{X_0, X_1, X_2, X_3\}$. The assignment statement $X := 4$ "Writes" the value 4 to X. The assignment statement $X_2 := 1$ "writes" the value 1 to X_2.

Definition: Consider a program fragment that replaces an assignment statement S of the implemented program. We say that the program fragment *Reads* (respectively, *Writes*) each higher-level variable that is Read (respectively, Written) by the assignment statement S; the set of all such higher-level variables comprise the *Read Set* (respectively, *Write Set*) of the program fragment. A program fragment is only allowed to access the lower-level variables that correspond to the higher-level variables in its Read Set and Write Set. □

Consider, for example, the program fragment for $X := 4$ given earlier. The Write Set for the program fragment consists of the single variable X, while the Read Set for the program fragment is empty. Note that the program fragment does not actually write X, but rather writes the corresponding lower-level variables X_0, X_1, X_2, and X_3; variable X exists only on a logical basis in the implementation.

Each execution of a program fragment has the effect of Reading a value from each variable in the Read Set of the program fragment, and Writing a value to each variable in the Write Set of the program fragment. The value "Read from" or "Written to" a given higher-level variable can be defined formally by specifying a function over the values read from and written to the lower-level variables that correspond to that higher-level variable. For instance, in the example considered at the beginning of this section, "the value Written to X" is defined by the function $v_0 + 2v_1 + 4v_2 + 8v_3$, where each v_i is the value written to X_i. Thus, associated with each program fragment is a set of functions, one for each higher-level variable in the program fragment's Read Set and Write Set. Note that if a variable appears in both the Read Set and Write Set of a program fragment, then two functions must be specified for that variable, one that defines the value Read from that variable, and another that defines the value Written to that variable.

We now define several concepts that are needed in order to formally define what it means for an implementation to be wait-free, sequentially correct, and serializable. The following definitions apply to any concurrent program.

Definition: A *state* is an assignment of values to the variables of the program. (Note that each process's program counter is considered to be a private variable of that process.) One or more states are designated as *initial states*. □

Definition: An *event* is an execution of a statement of the program. (In the case of a **do** or **if** statement, "execution of a statement" means the evaluation of each guard in the statement's set of guards, and the subsequent transfer of control.) □

Definition: Let t and u be any two states. If u is reached from t via the occurrence of event e, then we say that e is *enabled* at state t and we write $t \xrightarrow{e} u$. □

Definition: A *history* is a sequence $t_0 \xrightarrow{e_0} t_1 \xrightarrow{e_1} \cdots$. Unless noted otherwise, the first state t_0 is assumed to be an initial state. □

A given statement may be executed many times in a history. Each such statement execution corresponds to a distinct event.

Definition: Event e *precedes* another event f in a history iff e occurs before f in the history. □

The remaining definitions apply to a given implementation.

Definition: Let S_0; **do** $B_1 \rightarrow S_1 \;[\!]\; \cdots \;[\!]\; B_N \rightarrow S_N$ **od** be a program fragment of the implementation that replaces some assignment statement of the implemented program. Let h be a history of the implementation. The set of events in h corresponding to an execution of the program fragment — starting with an event that occurs as a result of executing statement S_0, and followed by all events that subsequently occur as a result of executing the statements in **do** $B_1 \rightarrow S_1 \;[\!]\; \cdots \;[\!]\; B_N \rightarrow S_N$ **od** — is called an *operation*. □

Definition: An operation p *precedes* another operation q in a history iff each event of p precedes all events of q. □

Observe that the precedes relation is an irreflexive total order on events and an irreflexive partial order on operations.

We now formally define the notions of wait-freedom, sequential correctness, and serializability. These concepts are defined with respect to the histories of the implementation.

Definition: A history of the implementation is *wait-free* iff each operation in the history consists of a finite number of events. □

In defining sequential correctness and serializability, it is sufficient to consider only "well-formed" histories. In a well-formed history, the values Read from and Written to the higher-level variables by each operation are well-defined.

Definition: A history of the implementation is *well-formed* iff each operation in the history is complete, i.e., the execution of the corresponding program fragment terminates. □

Definition: Let p be an operation in a well-formed history of the implementation, and let $x_1, \ldots, x_K := E_1, \ldots, E_K$ denote the assignment statement of the implemented program that corresponds to p. Let v_i denote the value Written to x_i by p, and let V_j denote the value obtained by replacing each variable appearing in E_j by the value Read from that variable by p. We say that p is *sequentially correct* iff the condition $v_i = V_i$ holds for each i. A well-formed history of the implementation is *sequentially correct* iff each operation in the history is sequentially correct. □

Next, we define what it means for a history of the implementation to be serializable. According to this definition, if several operations are executed concurrently, then the net effect should be equivalent to some serial order. Our definition of serializability is similar to the definition of linearizability given by Herlihy and Wing in [15].

Definition: Let h be a well-formed history of the implementation. History h is *serializable* iff the precedence relation on operations (which is a partial order) can be extended[3] to a total order $\sqsubset$ where for each operation p in h and each variable x that is Read by p, the value Read by p from x is the same as the last-Written value according to $\sqsubset$. More specifically,

- if there exists an operation q that Writes x such that $q \sqsubset p \wedge \neg(\exists q' : q'$ Writes $x : q \sqsubset q' \sqsubset p)$, then the value Read by p from x is the same as the value Written by q to x;
- if no such q exists, then the value Read by p from x is the same as the initial value of x as defined in the implemented program. □

We are now in a position to formally state the correctness condition for an implementation.

Definition: An implementation is *correct* iff each of its histories is wait-free, and each of its well-formed histories is sequentially correct and serializable. □

5 Necessity Results

In this section, we establish the necessity of Exact-Writing and Single-Phase. In particular, we show that for each of these constraints there exists a nonatomic program violating that constraint, but satisfying the other two constraints. This implies that if we remove either of these two constraints from our set of constraints, then we can no longer make the claim that any program satisfying all of our atomicity constraints is atomic.

The above-mentioned impossibility results are proved by considering the problem of two-process leader election; we call our version of this problem the *binary election problem*. We first show that this problem cannot be solved by a primitive program. We then give two programs that solve this problem: the first program violates Exact-Writing, and the second program violates Single-Phase. It follows, then, that neither of these programs can be implemented by a primitive program.

We now define the binary election problem.

[3] A relation R over a set S *extends* another relation R' over S iff for each a and b in S, $aR'b \Rightarrow aRb$.

shared var $X, Y, Z : 0..2$

initialization $X = Y = Z = 0$

process P	**process** Q
private var	**private var**
$electp$: **boolean** /* decision variable */	$electq$: **boolean** /* decision variable */
begin	**begin**
0: $X,\ Y := 1,\ 1$;	2: $Y,\ Z := 2,\ 2$;
1: $electp := ((Y,\ Z) = (1,\ 2))$	3: $electq := ((X,\ Y) = (1,\ 2))$
end	**end**

Figure 2: Proof of Theorem 4.

Binary Election Problem: We are required to construct a program consisting of two processes. Each process has a private boolean *decision variable* that it writes only once. The program is required to satisfy the following three conditions.

- *Wait-Freedom*: All histories of the program are finite.
- *Equity*: For each process, there exists a history with a final state in which the value of that process's decision variable is true.
- *Validity*: In the final state of every history, the value of one of the decision variables is true and the value of the other decision variable is false. □

In [5], we proved that the binary election problem cannot be solved by any program satisfying constraints (4) and (5) in the definition of primitive programs. In particular, we showed that if such a program satisfies both Equity and Validity, then it has an infinite history, violating Wait-Freedom. Similar proofs have been obtained independently by Chor, Israeli, and Li [11] and by Herlihy [14]. This establishes the following theorem.

Theorem 3: The binary election problem cannot be solved by a primitive program. □

We now show that the binary election problem can be solved if either Exact-Writing or Single-Phase is violated.

Theorem 4: The binary election problem can be solved by a program that violates Exact-Writing.

Proof: Consider the program given in Figure 2. Statement 0 of process P and statement 2 of process Q together constitute a violation of Exact-Writing. In particular, it is impossible to partition the variables X, Y, and Z into classes such that each of these two statements writes every variable from one class and no variable from any other class. This program satisfies Wait-Freedom because it is loop-free. To see that the program satisfies Equity and Validity, consider Figure 3. This figure depicts the possible values of (X, Y, Z); each arrow is labeled by the statement that causes the state

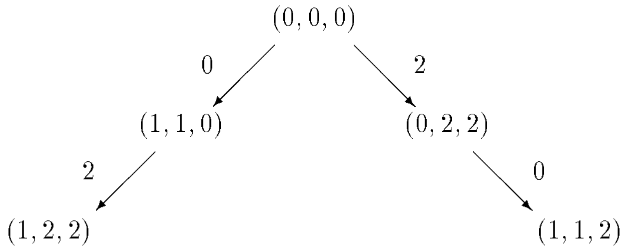

Figure 3: Possible values of (X, Y, Z).

shared var X : **boolean**

process P
private var
  electp : **boolean** /$*$ decision variable $*$/
begin
  $X,\ electp := \neg X,\ X$
end

process Q
private var
  electq : **boolean** /$*$ decision variable $*$/
begin
  $X,\ electq := \neg X,\ X$
end

Figure 4: Proof of Theorem 5.

change. Based on this figure, we conclude that either $(Y, Z) = (1, 2)$ when statement 1 is executed by P and $(X, Y) \neq (1, 2)$ when statement 3 is executed by Q, or vice versa. This implies that Equity and Validity are satisfied. $\Box$

Theorem 5: The binary election problem can be solved by a program that violates Single-Phase.

Proof: Consider the program given in Figure 4. Note that the assignment statement of each process both reads and writes the shared variable X and hence violates Single-Phase. It is easy to see that this program solves the binary election problem. $\Box$

6 Sufficiency Results

In this section, we discuss the proof strategy for establishing the sufficiency of our atomicity constraints. We prove that any program satisfying the three atomicity constraints is atomic; that is, such a program can be implemented by a primitive program (as described in Section 4). The implementation is facilitated by considering a shared data object, called a *composite register* [2, 3]. A composite register is an array-like variable that consists of a number of components. An operation of a composite register either writes a value to one of the components, or reads the values of all of the components. For convenience, we designate the structure of a given composite register by

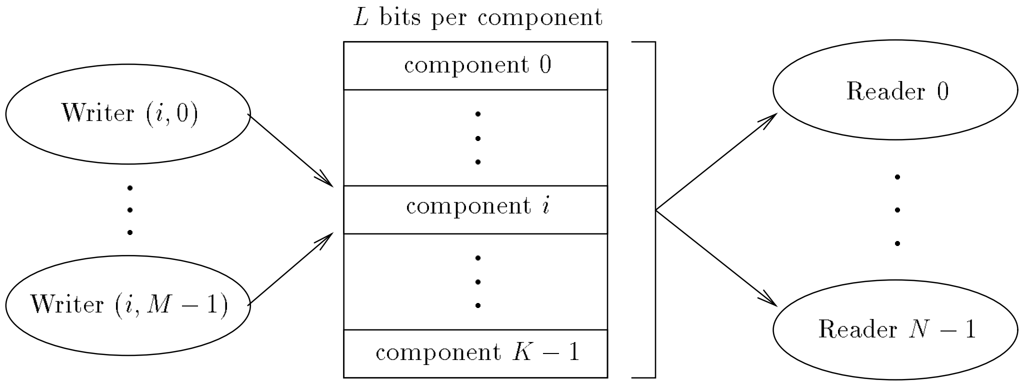

Figure 5: $K/L/M/N$ composite register structure.

a four-tuple $K/L/M/N$, where K is the number of components, L is the number of bits per component, M is the number of writers per component, and N is the number of readers. Note that, because integer variables are allowed in our definition of a program, L could be infinite. The structure of a $K/L/M/N$ composite register is shown in Figure 5. This figure only depicts the writers for component i.

We now present two theorems that reduce our proof obligation in establishing the sufficiency of our atomicity constraints to one involving only composite registers. First, we show that if the processes of a program communicate only by means of 1/1/1/1 composite registers, then that program is primitive. Second, we show that any program satisfying our atomicity constraints can be implemented by a program in which processes communicate only by means of $K/L/M/N$ composite registers. It follows from these two theorems that in order to establish the sufficiency of our atomicity constraints, it suffices to prove that a $K/L/M/N$ composite register can be constructed from 1/1/1/1 composite registers.

Theorem 6: If each shared variable of a program is a 1/1/1/1 composite register, then that program is primitive.

Proof: A 1/1/1/1 composite register is a single-reader, single-writer, shared bit. Recall that a program is primitive if its processes communicate only by means of such bits. Thus, if each shared variable of a concurrent program is a 1/1/1/1 composite register, then that program satisfies the six constraints in the definition of primitive programs. □

Theorem 7: Any program satisfying our proposed atomicity constraints can be implemented by a program in which each shared variable is a component of a $K/L/M/N$ composite register.

Proof Sketch: Consider a program A that satisfies the three atomicity constraints: Exact-Writing, Single-Phase, and Finite-Reading. Program A can be implemented by a program B in which no assignment statement both reads and writes shared variables. In particular, suppose that the following assignment statement appears in some process P of A.

$$x_1, \ \ldots, \ x_N := E_1, \ \ldots, \ E_N$$

If this statement both reads a shared variable and writes a shared variable, then, by Single-Phase, it reads no shared variable that is written by a process other than P. This implies that we can replace this statement by the two statements

$$v_1, \ \ldots, \ v_N := E_1, \ \ldots, \ E_N;$$
$$x_1, \ \ldots, \ x_N := v_1, \ \ldots, \ v_N$$

where each v_i is a new private variable of process P. By similarly replacing each assignment statement of A that both reads and writes shared variables, we obtain program B.

Now, we show that B can be implemented by a program C in which each assignment statement writes at most one shared variable. Consider an assignment statement of B that writes several shared variables $Z_1, \ \ldots, \ Z_K$. By Exact-Writing, each Z_i is an unsubscripted variable. Furthermore, if an assignment statement writes some Z_i, then it writes each Z_i and no other shared variable. We can therefore combine $Z_1, \ \ldots, \ Z_K$ into a new "compound" variable $ZNEW$ as follows:

$$ZNEW : \ \mathbf{record} \ Z_1 : type_1; \ \cdots \ Z_K : type_K \ \mathbf{end}$$

where $type_i$ is the type of Z_i in B. (Note that $ZNEW$ is really just an integer or subrange variable. We are using the **record** type only as "syntactic sugar" to indicate that the bits of $ZNEW$ are partitioned into various fields.) Each assignment statement of B that writes the K variables $Z_1, \ \ldots, \ Z_K$ is replaced by an assignment statement that writes the single variable $ZNEW$. We also have to replace each assignment statement that reads some Z_i. For example, the statement

$$a, \ b := Z_1 + Z_2, \ Z_3 + Z_4$$

(where a and b are private variables) would be replaced by the statement

$$a, \ b := ZNEW.Z_1 + ZNEW.Z_2, \ ZNEW.Z_3 + ZNEW.Z_4$$

Program C is obtained by making similar replacements for each class of shared variables given by Exact-Writing.

From the above construction, if an assignment statement of program C accesses a shared variable, then it either writes a single shared variable or reads one or more shared variables. Moreover, by the Finite-Reading constraint, if an assignment statement of C accesses an element of an infinite shared array, then it either reads that element or writes that element, and accesses no other shared variable.

Based upon these observations, we now show that C can be implemented by a program in which each shared variable is a component of a composite register. The implementation is obtained as follows: each element of an infinite shared array in C is replaced by a composite register with only one component; the remaining shared variables of C (other than elements of infinite shared arrays) are replaced by a single composite register, with each variable corresponding to one component of the register. From the construction of program C, it follows that each assignment statement can be implemented by either a write operation of one component of a composite register, or a read operation of all components of a composite register. $\square$

Example: We show that the example program considered in Section 3 can be implemented by a

```
type Ctype = record   VW : integer;   X : 1..N;   YZ : boolean   end
shared var C : array[1..2] of Ctype /* composite register */

initialization (C[2].VW = default value) ∧ (C[1].YZ = C[2].YZ)

process CONTROLLER
private var request, reply : integer;
            dev : 1..N;
            pending, y : boolean;
            c : array[1..2] of Ctype
begin
    do true → read c := C;
              reply, pending := c[2].VW, (c[1].YZ ≠ c[2].YZ);
              if ¬pending → consume reply;   produce next request, dev;
                            read c := C;   y := ¬c[1].YZ;
                            write C[1] := (request, dev, y)
              ▯ pending → skip
              fi
    od
end

process DEVICE(j : 1..N)
private var request, reply : integer;
            dev : 1..N;
            pending, z : boolean;
            c : array[1..2] of Ctype
begin
    do true → read c := C;
              request, dev, pending := c[1].VW, c[1].X, (c[1].YZ ≠ c[2].YZ);
              if pending ∧ dev = j → consume request;   produce reply;
                                     read c := C;   z := ¬c[2].YZ;
                                     write C[2] := (reply, 1, z)
              ▯ ¬pending ∨ dev ≠ j → skip
              fi
    od
end
```

Figure 6: Example program, revisited.

program with processes that communicate only via a single composite register. The corresponding program is shown in Figure 6. There is a single composite register C that replaces the five shared variables V, W, X, Y, and Z of the original program. C has two components, the first of which is written by the $CONTROLLER$, and the second of which is written by the $DEVICE$s.

To explain the structure of C, we return to the original program given in Figure 1. As suggested by Exact-Writing, we partition the five shared variables of this program into two classes $\{V, X, Y\}$ and $\{W, Z\}$. Each of these classes corresponds to one component of C: the class $\{V, X, Y\}$ corresponds to the first component of C, namely $C[1]$, and the class $\{W, Z\}$ corresponds to the second component of C, namely $C[2]$. Note that the class $\{V, X, Y\}$ contains an integer variable, a variable of type $1..N$, and a boolean variable, while the class $\{W, Z\}$ contains an integer variable and a boolean variable. To accommodate both classes, we define each component of C to include a field VW of type integer, a field X of type $1..N$, and a field YZ of type boolean. The correspondence between the shared variables of the original program and the composite register C is then as follows: variable V corresponds to $C[1].VW$, variable W corresponds to $C[2].VW$, variable X corresponds to $C[1].X$, variable Y corresponds to $C[1].YZ$, and variable Z corresponds to $C[2].YZ$. Note that $C[2].X$ is unused.

Using our tuple notation, C would be classified as a $2/\infty/N/N+1$ composite register: it has two components; each component consists of an infinite number of bits (since each VW field is an integer); each component can be written by N processes (although the $CONTROLLER$ actually only writes the first component and the N $DEVICE$s the second); and all $N+1$ processes can read C. Note that in Figure 6 we have annotated the program with the keywords **read** and **write** to emphasize those assignment statements that access the composite register C. This convention will also be adopted in the remainder of the paper. □

It follows from these two theorems that in order to establish the sufficiency of our atomicity constraints, our proof obligation is reduced to the following.

Proof Obligation: Show that a $K/L/M/N$ composite register can be implemented in terms of $1/1/1/1$ composite registers. □

Fortunately, this proof obligation can be divided into a series of steps, one for each of the parameters K, L, M, and N. Constructions for these steps can be found in the Ph.D. dissertation of the first author [4], as well as in the literature. The steps involved in constructing a $K/L/M/N$ composite register from $1/1/1/1$ composite registers are illustrated in Figure 7. Each arrow in this figure is labeled by a reference to the paper(s) in which the corresponding register construction(s) appear.

One of the steps shown in Figure 7 is considered in detail in Section 8. In particular, we present a construction of a $1/L/1/1$ composite register from $1/1/1/1$ composite registers. This construction is included to give the reader (of this paper) some idea as to the structure of a composite register construction. Our $1/L/1/1$ construction is the first that we know of that explicitly addresses the case in which L is allowed to be infinite.

In proving that a composite register construction is correct, the principal difficulty that arises is

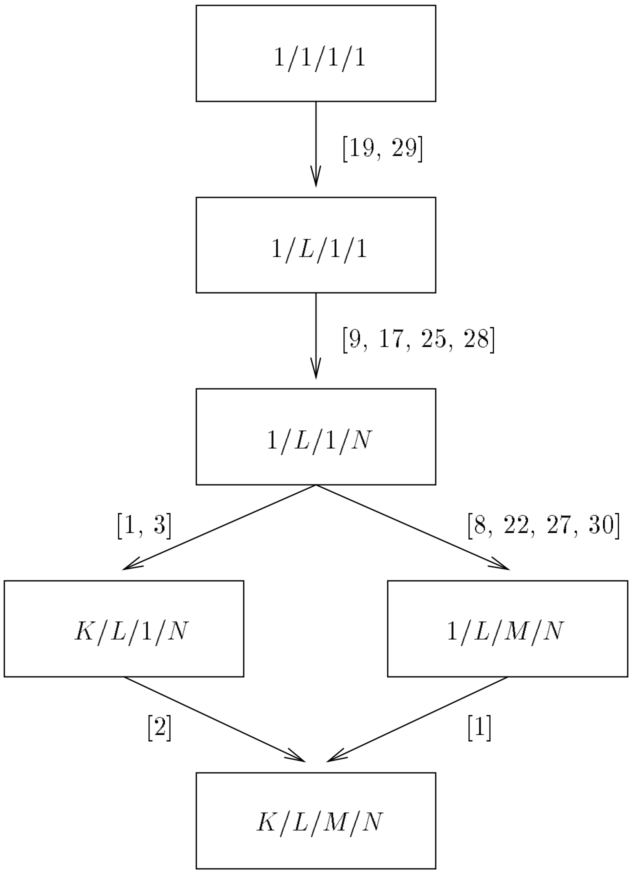

Figure 7: Constructing a $K/L/M/N$ composite register from $1/1/1/1$ composite registers.

that of establishing that all well-formed histories of the construction are serializable. The definition of serializability given in Section 4, while intuitive, is rather difficult to use directly. In the next section, we present a lemma that gives a set of conditions that are sufficient for establishing that a well-formed history of a construction is serializable. Intuitively, a history is serializable if each operation in the history can be shrunk to a point; that is, there exists a point between the first and last events of each operation at which the operation appears to take effect. For this reason, this lemma is referred to as the "Shrinking Lemma."

7 Shrinking Lemma

Before presenting the Shrinking Lemma, we first describe the basic structure of a composite register construction. We define a construction in a rather abstract way: we focus only on the Read and Write operations of the constructed register, while ignoring the processes that invoke these operations. More specifically, a construction consists of a set of Writer procedures and a set of Reader procedures that communicate via a set of lower-level variables. A Writer procedure is invoked by a process in order to Write a value to a component of the constructed composite register. A Reader procedure is invoked by a process in order to Read the values of all of the components of the constructed composite register. Each Writer procedure has one input parameter indicating the value to be Written; each Reader procedure has one output parameter for each component of the constructed register.

The Writer and Reader procedures of a construction correspond to the program fragments introduced in Section 4. The Read Set of a Writer procedure consists of the single input parameter of that procedure, while its Write Set consists of the single component Written by that procedure. The Read Set of a Reader procedure includes all of the components of the constructed register, while its Write Set includes all of its output parameters. In a history of a construction, the Writer and Reader procedures are invoked in an arbitrary manner.

The correctness condition for a construction is the same as defined in Section 4. That is, a construction is correct iff each of its histories is wait-free, and each of its well-formed histories is sequentially correct and serializable. Establishing that a history is wait-free and sequentially correct is usually straightforward. The major difficultly lies in establishing that a history is serializable, so it is this requirement that we will concentrate on here.

It is rather difficult to prove that a history of a composite register construction is serializable by appealing directly to the definition of serializability given in Section 4. The Shrinking Lemma, given below, reduces our proof obligation to five distinct conditions. Before stating this lemma, we first introduce some terminology.

Definition: A Write operation of component k of the constructed composite register, where $0 \leq k < K$, is called a *k-Write operation*. □

In order to avoid special cases when proving the correctness of a construction, we make the following assumption concerning the initial Write operations.

Initial Writes: For each k, where $0 \leq k < K$, there exists a k-Write operation that precedes each other k-Write operation and all Read operations. □

Shrinking Lemma: A well-formed history h of a $K/L/M/N$ composite register construction is serializable if for each k, where $0 \leq k < K$, there exists a function ϕ_k that maps every Read operation and k-Write operation in h to some natural number, such that the following five conditions hold.

- *Uniqueness*: For each pair of distinct k-Write operations v and w in h, $\phi_k(v) \neq \phi_k(w)$. Furthermore, if v precedes w, then $\phi_k(v) < \phi_k(w)$.

- *Integrity*: For each Read operation r in h, and for each k in the range $0 \leq k < K$, there exists a k-Write operation w in h such that $\phi_k(r) = \phi_k(w)$. Furthermore, the value Read by r for component k is the same as the value Written by w.

- *Proximity*: For each Read operation r in h and each k-Write operation w in h, if r precedes w then $\phi_k(r) < \phi_k(w)$, and if w precedes r then $\phi_k(w) \leq \phi_k(r)$.

- *Read Precedence*: For each pair of Read operations r and s in h, if $(\exists k :: \phi_k(r) < \phi_k(s))$ or if r precedes s, then $(\forall k :: \phi_k(r) \leq \phi_k(s))$.

- *Write Precedence*: For each Read operation r in h, and each j-Write operation v and k-Write operation w in h, where $0 \le j < K$ and $0 \le k < K$, if v precedes w and $\phi_k(w) \le \phi_k(r)$, then $\phi_j(v) \le \phi_j(r)$. □

Uniqueness totally orders the Write operations on a given component in accordance with the partial precedence ordering defined by h. According to Integrity, the value Read by a Read operation for a given component must equal the value Written to that component by some Write operation. This condition prohibits a Read operation from returning a predetermined value for some component. Proximity ensures that a Read operation does not return a value from the "future," or one from the "far past" that has subsequently been "overwritten" (i.e., each value Read by a Read operation must have been Written by a Write operation in close proximity). Read Precedence disallows two Read operations from obtaining inconsistent "snapshots" of the constructed register. Write Precedence orders Write operations of one component with respect to Write operations of another component. Note that for $K = 1$, Write Precedence is a direct consequence of Uniqueness. Conditions similar to Integrity, Proximity, and Read Precedence have been used elsewhere as a correctness condition for atomic register constructions; see, for example, the Integrity, Safety, and Precedence conditions of [28], and Proposition 3 of [19]. An atomic register is a special case of a composite register in which there is only one component.

The correctness proof for the Shrinking Lemma is given in [4]. The proof is somewhat tedious, but is not hard. The proof strategy is as follows. We first augment the precedence relation on operations in history h by adding pairs of operations. These added pairs of operations are defined based on the five conditions of the lemma. We then show that the resulting relation is an irreflexive partial order, i.e., it is irreflexive and transitive. Finally, we show that any extension of this relation to an irreflexive total order satisfies the conditions in the definition of serializability given in Section 4.

8 $1/L/1/1$ Construction

In this section, we establish that a $1/L/1/1$ composite register can be constructed from $1/1/1/1$ composite registers (even if L is infinite). For brevity, we only describe the construction informally here; a formal correctness proof is given in [4].

The construction is depicted in Figure 8. (We assume that any arithmetic expression involving L in this figure is replaced by ∞ for the case in which L is infinite.) We begin by giving a brief description of how the construction works; a more detailed description is given below. The Reader and the Writer communicate by means of four "buffers." Each buffer is an array of bits. The Writer Writes a value to the constructed register by copying the binary representation of this value into one of the four buffers. The Reader Reads a value from the constructed register by scanning one of the four buffers. To ensure that the Reader obtains a correct value, the Reader and the Writer are coordinated so that they never access the same buffer at the same time. This technique has been employed in several other similar constructions; see, for example, Burns and Peterson [10], Kirousis et al. [17], and Tromp [29].

The shared variables used in the construction are as follows.

type $Ytype$ = **array**$[0..2L]$ **of** $0..1$
shared var Y : **array**$[0..1][0..1]$ **of** $Ytype$;
Z : **array**$[0..1]$ **of** $0..1$;
$WP, RP : 0..1$

initialization $WP = RP \ \wedge\ (\forall i, j : 0 \le i \le 1 \ \wedge\ 0 \le j \le 1 : (\exists k : k \ge 0 : Y[i,j][2k] = 0))$

procedure $Reader$ **returns** $valtype$
private var
  $val : valtype$;
  $ralt, rp, x, y : 0..1$;
  $k : 0..L+1$
initialization
  $rp = RP$
begin
  read $rp := WP$;
  write $RP := rp$;
  read $ralt := Z[rp]$;
  $val,\ k := 0,\ 0$;
  do $k \le L\ \rightarrow$
    read $x := Y[rp, ralt][2k]$;
    if $x = 0\ \rightarrow$ **exit** $[\!]$ $x \ne 0\ \rightarrow$ **skip fi**;
    read $y := Y[rp, ralt][2k+1]$;
    $val,\ k := val + y \cdot 2^k, k+1$
  od;
  return(val)
end

procedure $Writer(val : valtype)$
private var
  $d, wp, walt : 0..1$;
  $n : 1..L$;
  $k : 0..L$
initialization
  $d = wp = WP$
begin
  read $d := RP$;
  $wp,\ walt,\ n := d \oplus 1,\ walt \oplus 1,$ number of bits in val;
  write $Y[wp, walt][2n] := 0$;
  $k := 0$;
  do $k < n\ \rightarrow$
    write $Y[wp, walt][2k] := 1$;
    write $Y[wp, walt][2k+1] := val[k]$;
    $k := k+1$
  od;
  write $Z[wp] := walt$;
  write $WP := wp$
end

Figure 8: $1/L/1/1$ construction.

$Y[i,j]$: A buffer that is used to store the binary representation of a value as Written by a Write operation. There are four such buffers since i and j each range over $\{0,1\}$. Each buffer is an array of bits written by the Writer and read by the Reader. If L is finite, then $Y[i,j]$ has $2L+1$ elements, and if L is infinite, then $Y[i,j]$ has an infinite number of elements. (Note that Y is really a three-dimensional array. However, in describing how the construction works, we find it convenient to view Y as a two-dimensional array, where each element is itself an array.)

The following shared variables are used to ensure that the Reader and Writer never access the same buffer at the same time. This is described in detail below.

$Z[i]$: A bit that is written by the Writer and read by the Reader. There are two such bits since i ranges over $\{0,1\}$.

WP: A modulo-2 "write pointer" that is written by the Writer and read by the Reader. (We use $\oplus$ to denote modulo-2 addition.)

RP: A modulo-2 "read pointer" that is written by the Reader and read by the Writer.

To simplify the Reader and Writer procedures in Figure 8, we assume that the private variables of each procedure retain their values between invocations.

A Write operation Writes a new value to the constructed register by copying the binary representation of this value into the buffer $Y[wp, walt]$. Thus, wp and $walt$ together constitute a two-bit "pointer" to one of the four shared buffers. The value of wp is obtained by first reading RP and then incrementing the value read. The value of $walt$ is obtained by incrementing the value assigned to $walt$ by the previous Write operation. After updating $Y[wp, walt]$, the value of $walt$ is made available to the Reader by writing it to $Z[wp]$, and the value of wp is made available to the Reader by writing it to WP.

Each pair of bits $(Y[wp, walt][2k], Y[wp, walt][2k+1])$, where $k \geq 0$, encodes a single bit of the Written value. The pair $(1,0)$ encodes a bit with value 0 and the pair $(1,1)$ encodes a bit with value 1. The pair $(0,-)$ is used to indicate the end of a string of bits. Note that such a string can be arbitrarily long since L may be infinite — hence the need for the encoding.

A Read operation Reads a value from the constructed register by scanning one of the four shared buffers. A Read operation scans the buffer that is pointed to by the pair of bits rp and $ralt$. The value of rp is obtained by reading WP and the value of $ralt$ is obtained by reading $Z[rp]$. The value of rp is made available to the Writer by writing it to RP.

The correctness of this construction follows from the following fact (which is proved in [4]): if a Read operation executes its **do** statement while a Write operation executes its **do** statement, then $rp \neq wp$ or $ralt \neq walt$. This implies that no Write operation interferes with a Read operation as it reads from one of the four shared buffers.

The space complexity of the construction is determined by the number of shared 1/1/1/1 composite registers required. If L is infinite, then an infinite number of 1/1/1/1 registers are obviously required. So, assume that L is finite. Then, the complexity of each of the shared variables is as follows.

- $Y[i,j]$, where $0 \leq i,j \leq 1$, uses $2L+1$ bits.
- $Z[i]$, where $0 \leq i \leq 1$, uses 1 bit.
- WP uses 1 bit.
- RP uses 1 bit.

Therefore, the space complexity is $8L+8$. The lower bound to construct an L-bit register is $O(L)$. Thus, our construction is asymptotically optimal.

The time complexity of a $1/L/1/1$ construction is determined by the number of reads and writes of shared 1/1/1/1 registers required to Read and Write the constructed register. The time complexity of a Read in our $1/L/1/1$ construction is $O(L')$, where $L' < L$ is the size (in bits) of the value Read. The time complexity of a Write is $O(L')$, where $L' < L$ is the size (in bits) of the value Written.

9 Concluding Remarks

We have presented a new criterion for the atomicity of concurrent programs along with a simple method for checking whether a given program is atomic. This method consists of three constraints, which can often be checked based on the syntax of a particular program. The class of programs considered atomic according to our atomicity criterion is larger than the class considered atomic according to the traditional Owicki-Gries atomicity criterion.

Our atomicity criterion is motivated by the following observation: Interleaving semantics is a suitable abstraction only if we restrict our attention to programs with sufficiently simple actions. According to our criterion, an action is "sufficiently simple" if it can be implemented in a wait-free, serializable manner in terms of primitive actions. (In [13], Gouda argues that we should also restrict our attention to only certain kinds of assertions when reasoning about the behavior of a program under interleaving semantics. In particular, he considers a restricted notion of parallelism and argues that for interleaving semantics to be a suitable abstraction for this notion of parallelism, we should limit our attention to "parallelizable" assertions.)

Our notion of serializability is equivalent to the definition of linearizability given by Herlihy and Wing in [15], for the special case of implementing a program in which shared variables are accessed only by assignment statements. A similar notion has also been proposed by Misra [24]. In particular, Misra gives a set of axioms for constructing concurrent hardware registers, and shows that if a register satisfies these axioms, then concurrent accesses to the register can be viewed as being interleaved.

To establish the sufficiency of our atomicity constraints, we considered in Section 6 a shared data object called a composite register. The notion of a composite register is an extension of an atomic register, and was first introduced by Anderson in [2, 3]. A composite register is equivalent to the atomic snapshot primitive defined by Afek et al. in [1]. The notion of an atomic register was first defined by Lamport [19]. An atomic register is a composite register with only one component.

We should point out that it may be possible to relax the Exact-Writing and Single-Phase constraints somewhat, despite the necessity results proved in Section 5. For example, according to Exact-Writing, if two assignment statements in different processes write a common shared variable, then they write the same set of shared variables. In establishing the necessity of Exact-Writing, we considered in Theorem 4 a program in which the two assignment statements $X, Y := 1, 1$ and $Y, Z := 2, 2$ appeared in different processes. Each of these assignment statements writes one shared variable that is written by the other, and one that is not. This necessity proof leaves open the possibility of relaxing Exact-Writing to allow an assignment statement of one process to write a subset of the variables that are written by an assignment of another process; for example, $X, Y, Z := 1, 1, 1$ and $Y, Z := 2, 2$.

Recent results concerning "pseudo read-modify-write" (PRMW) operations show that Single-Phase can also be relaxed somewhat. The PRMW operation takes its name from the classical read-modify-write (RMW) operation as defined in [18]; the RMW operation has the form "$temp, X := X, f(X)$," where X is a shared variable, $temp$ a private variable, and f a function. Executing this operation has the effect of modifying the value of X according to function f, and returning the original value of X in $temp$. The PRMW operation has the form "$X := f(X)$," and differs from the RMW operation in that the value of X is not returned. It is shown in [6] and [7] that any shared data object that can either be read, written, or modified by a commutative PRMW operation can be implemented without waiting from atomic registers. This work shows that under certain conditions it is permissible to allow an assignment statement of one process to both read and write a shared variable that is written by another process.

It may also be possible to relax the atomicity constraints based upon the semantics of a particular program. For instance, consider a program in which one process executes the statement $X := Y$ and another process executes the statement $Y := X$. This program violates Single-Phase. However, if the two processes are synchronized in such a way so that these two statements are never enabled at the same time, then it may be reasonable to consider both statements to be atomic. We have tried as much as possible to avoid the need for analyzing the semantics of a program when appealing to our atomicity constraints.

As mentioned earlier, the version of Exact-Writing considered in this paper is overly restrictive because it allows an assignment statement to write several shared variables only if all such variables are unsubscripted. In [4], a more permissive version of Exact-Writing is considered that allows an assignment statement to write several subscripted variables. This version of Exact-Writing is similar to the one given in this paper, except for the partitioning of shared variables into classes. In [4], a class is allowed to consist of several shared variables in the following cases.

- A class may consist of a number of unsubscripted variables.
- A class may consist of all elements of a finite array.
- If $A, B, \ldots, C$ are single-dimensional arrays whose subscripts have the same range, then their elements can be partitioned by subscript, i.e., $A[i], B[i], \ldots, C[i]$ is a class for each i. Multi-dimensional arrays may be similarly partitioned.

The above version of Exact-Writing is still, in fact, more restrictive than is necessary. However, there is a tradeoff here: the more permissive we are concerning classes, the more difficult it will be to establish the sufficiency of our atomicity constraints.

We leave the question of whether Finite-Reading is necessary as an open problem. To prove that Finite-Reading is necessary, we are required to demonstrate the existence of a nonatomic program that violates Finite-Reading, but satisfies Exact-Writing and Single-Phase. On the other hand, to prove that Finite-Reading is not necessary (i.e., can be removed from our proposed set of atomicity constraints), we are required to show that any program satisfying only Exact-Writing and Single-Phase can be implemented in terms of single-reader, single-writer, shared bits. Note that the approach that we took in Theorem 2 will not work in this case. The principal difficulty that arises is illustrated by the following example. Consider a concurrent program with a process that includes the following program fragment, where the infinite array $X[0..\infty]$ is shared.

$i := 0;$
do $true \;\rightarrow\; a[i],\ b[i],\ i := X[i],\ X[i+1],\ i+1$ **od**

This program violates Finite-Reading because each assignment statement $a[i], b[i],\ i := X[i],\ X[i+1],\ i+1$, where $i \geq 0$, reads two elements of the infinite shared array $X[0..\infty]$. In order to implement the variables of this program in terms of composite registers, it is necessary to partition the elements of X into finite classes and implement each such class by using a single composite register. This partitioning is necessary because a composite register, by definition, consists of a finite number of components. (An atomic snapshot of an infinite number of components cannot be implemented in a wait-free manner.) However, it is impossible to partition the elements of X so that all shared variables accessed by each assignment statement come from the same class.

Acknowledgements: We would like to thank Ted Herman and Fred Schneider for their comments on an earlier draft of this paper. We would also like to thank the following members of the UT Distributed Systems Discussion Group for their careful reading of a draft of this paper: Yeturu Aahlad, Bryan Bayerdorffer, Ken Calvert, Al Carruth, Edgar Knapp, Jacob Kornerup, David Naumann, and J.R. Rao. We are also indebted to Zhou Chaochen and the referees for their helpful suggestions.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic Snapshots of Shared Memory," *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, pp. 1-14.

[2] J. Anderson, "Multiple-Writer Composite Registers," Technical Report TR.89.26, Department of Computer Sciences, University of Texas at Austin, September 1989.

[3] J. Anderson, "Composite Registers," *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, pp. 15-30.

[4] J. Anderson, *Atomicity of Concurrent Programs*, Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, 1990.

[5] J. Anderson and M. Gouda, "The Virtue of Patience: Concurrent Programming With and Without Waiting," Technical Report TR.90.23, Department of Computer Sciences, University of Texas at Austin, 1990.

[6] J. Anderson and B. Grošelj, "Pseudo Read-Modify-Write Operations: Bounded Wait-Free Implementations," *Proceedings of the Fifth International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science, Springer Verlag, to appear.

[7] J. Aspens and M. Herlihy, "Wait-Free Data Structures in the Asynchronous PRAM Model," *Proceedings of the Second Annual ACM Symposium on Parallel Architectures and Algorithms*, July, 1990.

[8] B. Bloom, "Constructing Two-Writer Atomic Registers," *IEEE Transactions on Computers*, Vol. 37, No. 12, December 1988, pp. 1506-1514.

[9] J. Burns and G. Peterson, "Constructing Multi-Reader Atomic Values from Non-Atomic Values," *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 222-231.

[10] J. Burns and G. Peterson, "Pure Buffers for Concurrent Reading While Writing," Technical Report GIT-ICS-87/17, School of Information and Computer Science, Georgia Institute of Technology, April 1987.

[11] B. Chor, A. Israeli, and M. Li, "On Processor Coordination Using Asynchronous Hardware," *Principles of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 86-97.

[12] E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[13] M. Gouda, "Serializable Programs, Parallelizable Assertions: A Basis for Interleaving," in *Beauty is Our Business*, eds. W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, Springer-Verlag, 1990, pp. 135-140.

[14] M. Herlihy, "Wait-Free Synchronization," *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, 1991, pp. 124-149.

[15] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, 1990, pp. 463-492.

[16] C.A.R. Hoare, "Towards a Theory of Parallel Programming," *Operating Systems Techniques*, eds. Hoare and Perott, Academic Press, New York, 1972.

[17] L. Kirousis, E. Kranakis, and P. Vitanyi, "Atomic Multireader Register," *Proceedings of the Second International Workshop on Distributed Computing*, Lecture Notes in Computer Science 312, pp. 278-296, Springer Verlag, 1987.

[18] C. Kruskal, L. Rudolph, M. Snir, "Efficient Synchronization on Multiprocessors with Shared Memory," *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 4, October 1988, pp. 579-601.

[19] L. Lamport, "On Interprocess Communication, Parts I and II," *Distributed Computing*, Vol. 1, 1986, pp. 77-101.

[20] L. Lamport, "*win* and *sin*: Predicate Transformers for Concurrency," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, 1990, pp. 396-428.

[21] L. Lamport and F. Schneider, "The Hoare Logic of CSP, and All That," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 2, April 1984, pp. 281-296.

[22] M. Li, J. Tromp, and P. Vitanyi, "How to Construct Wait-Free Variables," *Proceedings of International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science 372, pp. 488-505, Springer Verlag, 1989.

[23] Z. Manna and A. Pnueli, "Adequate Proof Principles for Invariance and Liveness Properities of Concurrent Programs," *Science of Computer Programming*, Vol. 4, 1984, pp. 257-289.

[24] J. Misra, "Axioms for Memory Access in Asynchronous Hardware Systems," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 1, January 1986, pp. 142-153.

[25] R. Newman-Wolfe, "A Protocol for Wait-Free, Atomic, Multi-Reader Shared Variables, *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 232-248.

[26] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica*, Vol. 6, 1976, pp. 319-340.

[27] G. Peterson and J. Burns, "Concurrent Reading While Writing II: The Multi-Writer case," *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, 1987.

[28] A. Singh, J. Anderson, and M. Gouda, "The Elusive Atomic Register, Revisited," *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 206-221.

[29] J. Tromp, "How to Construct an Atomic Variable," *Proceedings of the Third International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 392, pp. 292-302, Springer Verlag, 1989.

[30] P. Vitanyi and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware," *Proceedings of the 27th IEEE Symposium on the Foundations of Computer Science*, 1986, pp. 233-243.