The Granularity of Waiting
(Extended Abstract)

James H. Anderson[*] Jae-Heon Yang[*] Mohamed G. Gouda[†]

April 1992

Abstract

We examine the "granularity" of statements of the form "**await** $B \rightarrow S$", where B is a boolean expression over program variables and S is a multiple-assignment. We consider two classes of such statements to have the same granularity iff any statement of one class can be implemented without busy-waiting by using statements of the other class. Two key results are presented. First, we show that statements of the form "**await** $B \rightarrow S$" can be implemented without busy-waiting by using simpler statements of the form "**await** X", "$X := y$", and "$y := X$", where y is a private boolean variable and X is a shared singler-reader, multi-writer boolean variable. Second, we show that if busy-waiting is not allowed, then there is no general mechanism for implementing statements of the form "**await** B", where B is an N-writer expression, using only assignment statements and statements of the form "**await** C", where C is an $(N-1)$-writer expression. It follows from these results that the granularity of waiting depends primarily on the number of processes that may write each program variable. These results also show that, from a computational standpoint, operations that combine both waiting and assignment, such as the P semaphore primitive, are not fundamental.

Keywords: Atomicity, busy-waiting, conditional mutual exclusion, idle-waiting, implementations, linearizability, mutual exclusion, shared data, synchronization primitives.

1 Introduction

Atomic operations are commonly categorized by "granularity": an operation is said to be *fine-grained* if it can be easily implemented in terms of low-level machine instructions, and is said to be *coarse-grained* otherwise. Coarse-grained atomic operations are used when developing concurrent programs in a top-down fashion; under this approach, a program is first developed using coarse-grained operations, and then each coarse-grained operation is implemented by fine-grained ones.

[*] Department of Computer Science, The University of Maryland at College Park, College Park, Maryland 20742-3255. Work supported, in part, by NSF Contract CCR 9109497, and by the Center for Excellence in Space Data and Information Sciences. The first author was also supported by an award from the General Research Board at the University of Maryland. E-mail: jha@cs.umd.edu. Phone: (301) 405-2701.

[†] Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712-1188

In this paper, we consider the latter problem, i.e., that of implementing one kind of atomic operation in terms of another. Our specific goal is to determine the extent to which such implementations can be achieved without busy-waiting. This has been recognized as an important question for many years, as evidenced by the following quote taken from a paper written by Dijkstra in 1976 [8].

> To what extent the ideal "no unbounded repetitions in the individual programs" [busy-waiting] can be achieved in general — possibly by allowing certain *special* units of action to refer to more than one shared variable — is a question to which I don't know the answer at the moment of this writing.

The disadvantages of busy-waiting are twofold. First, programs with processes that busy-wait may suffer from performance degradation: a busy-waiting process not only wastes processor cycles, but also consumes memory bandwidth [10, 19]. Second, and perhaps more importantly, the use of busy-waiting often results in programs that are difficult to analyze and prove correct [5].

Recent work on wait-free synchronization has largely answered Dijkstra's question for the case of operations that only read or write shared variables; representative papers on wait-free synchronization include [1, 2, 4, 6, 11, 13, 15]. In this paper, we extend this work by considering conditional operations, i.e., operations with enabling conditions that involve shared variables. The P semaphore primitive is an example of such an operation: it consists of an assignment "$X := X - 1$", where X is shared, that may be executed only when the enabling condition "$X > 0$" holds. We represent conditional operations by means of statements of the form "**await** $B \rightarrow S$", where B is a boolean expression over program variables and S is a multiple-assignment. This statement can be executed only when its enabling expression B is true. It is atomically executed (when enabled) by performing its assignment S. We abbreviate such a statement as "**await** B" if its assignment is null, and as "S" if its enabling expression is identically true.

Because conditional operations may require processes to wait, wait-free implementations of them in general do not exist. Thus, we are left with a large gap in our understanding of the concept of "granularity". In this paper, we bridge this gap by considering the relative granularity of various classes of **await** statements. As suggested above, we consider two classes of such statements to have the same granularity iff any statement of one class can be implemented without busy-waiting by using statements of the other class. This notion of granularity extends that used in work on wait-free synchronization.

In the remainder of the paper, two key results are presented.

- First, we prove that statements of the form "**await** $B \rightarrow S$" can be implemented without busy-waiting by using simpler statements of the form "**await** X", "$X := y$", and "$y := X$", where y is a private boolean variable and X is a shared, single-reader, multi-writer boolean variable.[1] This result shows that, from a computational standpoint, operations that combine both waiting and assignment, such as the P semaphore primitive, are *not* fundamental.

- Second, we show that if busy-waiting is not allowed, then there is no general mechanism for implementing statements of the form "**await** B", where B is an N-writer expression (i.e., one whose value can be changed by N distinct processes) by using only assignment statements and statements of the form "**await** C", where C is an $(N-1)$-writer expression.

[1] An *m-reader, n-writer variable* can be read or waited on by m processes and can be written by n processes. For simplicity, we do not distinguish between reading and waiting when classifying variables in this way.

As intermediate steps in establishing the former result, we present solutions to two synchronization problems. The first is a solution to a new synchronization problem, defined here for the first time, called the *conditional mutual exclusion problem*. The second is a new solution to the mutual exclusion problem in which processes do not busy-wait and in which only single-reader, single-writer boolean variables are used.

It follows from the two results mentioned above that the granularity of waiting depends primarily on the number of processes that can write each shared variable. Other characteristics, such as the number of processes that may read or wait on each shared variable, the size of each shared variable, and the number of shared variables that can be accessed within a single statement, are not as important. Further, these results establish that Dijkstra's ideal, "no busy-waiting", can be realized by using "special units of action" of the form "**await** X ", "$X := y$", and "$y := X$", where y is a private boolean variable and X is a shared, single-reader boolean variable that can be written by any process.

The rest of this paper is organized as follows. In Section 2, we present our model of concurrent programs and define what it means to implement an **await** statement of one class by using **await** statements of another class. The results mentioned in the preceding paragraph are explained in more detail in Section 3 and are formally established in Sections 4 through 6. Concluding remarks appear in Section 7.

2 Concurrent Programs and Implementations

A *concurrent program* consists of a set of processes and a set of variables. A *process* is a sequential program consisting of labeled statements, and is specified using guarded commands [7] and **await** statements. 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 accessed by more than one process. Each process of a concurrent program has a special private variable called its *program counter*: the statement with label k in process p may be executed only when the value of the program counter of p equals k. To facilitate the presentation, we assume that shared variables appear only in **await** statements. For an example of the syntax we employ for programs, see Figure 2.

A program's semantics is defined by its set of "fair histories". As defined formally in the full paper, a *history* is a structure that represents a single execution of a program. In the usual way, we represent a history of a program as a sequence $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, where t_0 is an initial state of the program and $t_i \xrightarrow{s_i} t_{i+1}$ denotes the fact that state t_{i+1} is reached from state t_i via the execution of statement s_i. Informally, a history of a program is *fair* iff each statement of the program is either infinitely often disabled for execution in the history or is infinitely often executed in the history. Unless otherwise noted, we henceforth assume that all histories are fair.

When reasoning about the correctness of a concurrent program, safety properties are defined using invariants and progress properties are defined using leads-to assertions. An assertion B (over program variables) is an *invariant* of a program iff B holds in each state of every history of the program. We say that predicate B *leads-to* predicate C in a program, denoted $B \mapsto C$, iff for each history $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$ of the program, if B is true at some state t_i, then C is true at some state t_j where $j \geq i$.

As stated in the introduction, we consider two classes of **await** statements to have the same granularity iff any statement of one class can be implemented without busy-waiting by using statements of the other class. We define this notion of an implementation precisely by defining what it means to implement one *program* by another. Our notion of an implementation is defined with respect to programs because a given **await** statement's implementation may depend on the context in which that statement appears. If program P is implemented by program Q, then we refer to P as the *implemented program*, and Q as the *implementation*. (Presumably, P has "coarse-grained" **await** statements, whereas Q has "fine-grained" ones.)

In the full paper, we formally define the conditions required of an implementation. Informally, an implementation is obtained by replacing each **await** statement of the implemented program by a program fragment that has the same "effect" as that statement when executed in isolation. Such a program fragment is restricted to be free of unbounded busy-waiting loops. Although different program fragments in different processes may be executed concurrently (i.e., their statements may be interleaved), each program fragment must "appear" to be atomic; this condition is formalized by requiring all histories of the implementation to be *linearizable* [9].

One way to ensure linearizable execution is to use critical sections. This is the approach taken in most implementations presented in this paper. In such an implementation, each statement of the form "**await** $B \rightarrow S$" is implemented by executing the assignment S as a critical section. Observe that the critical section that implements S can be executed only when the enabling predicate B holds. This aspect of conditional synchronization is not taken into account in traditional synchronization paradigms such as the mutual exclusion problem. We will have more to say about this in Section 4.

3 Results

In this section, we outline the results presented in the remainder of the paper. As mentioned in the introduction, our most important contribution is to show that the granularity of waiting depends primarily on the number of processes that may write each program variable. This conclusion is based on two key results, which are given in Theorems 1 and 2 below. In these theorems, we consider programs called "k-primitive programs".

k-Primitive Programs: A program is *k-primitive* iff each of its **await** statements is either of the form "**await** X", "$X := y$", or "$y := X$", where y is a private boolean variable and X is a shared, single-reader, k-writer, boolean variable. □

We first consider a number of lemmas that are needed in order to establish Theorem 1.

Lemma 1: Any program can be implemented by a program in which each **await** statement is either of the form "**await** B" or "S". □

We establish this lemma in Section 4 by considering a variant of the mutual exclusion problem called the *conditional mutual exclusion problem*. In the conditional mutual exclusion problem, there is a predicate associated with each process that must be true when that process executes its critical section. This problem is motivated by our desire to implement statements of the form "**await** $B \rightarrow S$"

by using statements of the form "**await** B" and "S". Our solution to this problem shows that it is possible to implement any statement that combines both waiting and assignment in terms of statements that do not. The next two lemmas show that we can simplify **await** statements of the form "**await** B" and "S", respectively.

Lemma 2: Any program in which each **await** statement is either of the form "**await** B" or "S" can be implemented by a program in which each **await** statement is either of the form "**await** X" or "S", where X is a shared, single-reader, multi-writer boolean variable.

Proof Sketch: We use $B_1, \ldots, B_N$ to denote the enabling predicates of statements of the form "**await** B" appearing in the implemented program. The implementation is obtained by replacing each statement of the form "**await** B_k" by a statement of the form "**await** X_k", where X_k is a shared boolean variable that differs from any appearing in the implemented program; X_k is initially true iff predicate B_k is initially true. Each assignment "S" of the implemented program that may possibly modify B_k is modified to assign $X_k := B_k$. This ensures that $X_k = B_k$ is kept invariant for each k. □

Lemma 3: Any program in which each **await** statement is either of the form "**await** X" or "S", where X is a shared, single-reader, multi-writer boolean variable, can be implemented by a k-primitive program for some k.

Proof Sketch: In Section 5, we prove that the mutual exclusion problem can be solved without busy-waiting using only single-reader, single-writer, boolean variables. As shown in the full paper, it is straightforward to use this solution to the mutual exclusion problem to obtain a k-primitive implementation. The required implementation is obtained by first implementing each assignment "S" as a critical section and by then modifying the program so that only single-reader boolean variables are used. (The latter is easy to do since assignments of the implemented program are executed as critical sections.) □

The preceding three lemmas establish the following theorem.

Theorem 1: Any program can be implemented by a k-primitive program for some k. □

According to Theorem 1, any program can be "reduced" to one in which each **await** statement is as fine-grained as possible, with the exception of multi-writer variables. In Section 6, we prove that, in general, this "multi-writer barrier" cannot be crossed. In particular, we consider a variant of the termination detection problem in which an "observer" process detects the termination of two "worker" processes. We first show that this problem can be solved without busy-waiting if the observer is allowed to wait on an expression that may be modified by both workers. We then show that such a solution is impossible if the observer can wait on only one worker at a time. This result establishes the following theorem.

Theorem 2: There exists a program that cannot be implemented by any 1-primitive program. □

4 Conditional Mutual Exclusion

In this section, we define the conditional mutual exclusion problem. We then present a program that solves this problem in which processes do not busy-wait and in which only **await** statements of the form "**await** B" and "S" are used. Our solution to this problem is used in the proof of Lemma 1 in Section 3. In the conditional mutual exclusion problem, there are N processes, each of which has the following structure.

```
do true →
    Noncritical Section;
    Entry Section;
    Critical Section;
    Exit Section
od
```

Associated with each process i is an enabling predicate $B[i]$ that must be true when that process enters its critical section. An enabling predicate's value can be changed only by a process in its critical section. It is assumed that each critical section execution terminates. By contrast, a process is allowed to halt in its noncritical section. No variable appearing in any entry or exit section may be referred to in any noncritical section. Also, with the exception of enabling predicates, no such variable may be referred to in any critical section. Let $ES(i)$ ($CS(i)$) be a predicate that is true iff the value of process i's program counter equals a label of a statement appearing in its entry section (critical section). Let $BCS(i)$ be a predicate that is true iff the value of process i's program counter equals the label of the first statement in its critical section. (For simplicity, we assume that this statement is executed once per critical section execution.) Then, the requirements that must be satisfied by a program that solves this problem are as follows.

- *Mutual Exclusion*: $(\forall i, j : i \neq j :: CS(i) \;\Rightarrow\; \neg CS(j))$ is an invariant. Informally, at most one process can execute its critical section at a time.

- *Synchrony*: $(\forall i :: BCS(i) \;\Rightarrow\; B[i])$ is an invariant. Informally, when a process first enters its critical section, its enabling predicate is true.

- *Progress*: $(\forall i :: ES(i) \;\mapsto\; CS(i) \;\vee\; \neg B[i])$ holds. Informally, if a process is in its entry section and its enabling predicate continuously holds, then that process eventually executes its critical section.

We also require that each process in its exit section eventually enters its noncritical section; this requirement holds trivially for all solutions considered in this paper, so we will not consider it further. Observe that the conditional mutual exclusion problem reduces to the mutual exclusion problem when each process's enabling predicate is always identically true.

If busy-waiting is allowed, then it is straightforward to use a solution to the mutual exclusion problem to obtain a program that solves the conditional mutual exclusion problem. In particular, consider the program given in Figure 1, which is taken from [5]. In this program, ENTRY and EXIT denote entry and exit sections from an N-process solution to the mutual exclusion problem. In order to execute its critical section, process i repeatedly executes ENTRY and EXIT, checking $B[i]$ in between. The

```
process $i$
do $true$ $\rightarrow$
    Noncritical Section;
    ENTRY;
    do $\neg B[i]$ $\rightarrow$ EXIT; ENTRY od;
    Critical Section;
    EXIT
od
```

Figure 1: Using mutual exclusion to solve conditional mutual exclusion.

critical section is entered only if $B[i]$ is true; otherwise, EXIT and ENTRY are executed again. Note that when process i has executed ENTRY but not EXIT, it is effectively within its "mutual exclusion critical section".

In the mutual exclusion problem, a process gets to its critical section by establishing "priority" over other processes. In the conditional mutual exclusion problem, a process may have to relinquish and establish priority over other processes an unbounded number of times before executing its critical section. To see this, observe that the enabling predicate of a given process u may be repeatedly falsified and established by other processes; if u is in its entry section, then in the former case, u must relinquish priority over other processes, and in the latter case, u must again establish priority. It is this aspect of the conditional mutual exclusion problem that makes a solution without busy-waiting problematic.

A program that solves the conditional mutual exclusion problem without busy-waiting is given in Figure 2. This program is derived from Peterson's solution to the N-process mutual exclusion problem given in [18]. Processes "transit" through $N+1$ levels numbered from 0 to N. Starting from level N, processes compete to enter level 0. A process at level 0 executes its critical section. $Q[u]$ represents process u's current level, and $u.q$ is a private copy of $Q[u]$. The **await** statement shown in Figure 2 allows a process at level $j+1$ to enter level j only if there are at most j processes in levels 0 through j. Observe that, if process u's enabling predicate $B[u]$ is false, then process u's **await** statement is disabled.[2] $T[j]$ records the process that arrived last at level j. The **always** section in Figure 2 is used to define two expressions $C(u)$ and $D(u)$, which appear as shorthand in the program text; in the definition of these expressions, we implicitly assume that p, v, and w each range over $\{0, \ldots, N-1\}$. Roughly speaking, $C(u)$ enables process u to proceed when there is no process at level 0 and u is at the lowest numbered level among those processes whose enabling predicates hold. $D(u)$ enables process u to proceed if there is another process v that arrived later at process u's current level and that process is not at level 0.

The propositions that are needed to prove Mutual Exclusion and Synchrony are as follows. In these assertions, $i@\{S\}$ holds iff the program counter of process i equals some value in set S.

invariant $(\forall i : 0 \leq i < N :: i@\{5\} \Rightarrow B[i])$

[2] Although it is clear that the **await** statement of Peterson's algorithm must be modified to take the enabling predicates into account, choosing a correct modification out of the many possible alternatives is not trivial.

```
shared var    $Q,\ T$ : array$[0..N-1]$ of $0..N$;
              $B$     : array$[0..N-1]$ of boolean
initially     $(\forall\ i :: Q[i] = N \ \wedge\ T[i] = N)$
always        $C(u)\ \equiv\ (\forall\ p : p \neq u :: (B[p] \ \Rightarrow\ Q[p] > u.q)\ \wedge\ Q[p] \neq 0)$
              $D(u)\ \equiv\ (\exists\ v, w : v \neq u :: w = u.q\ \wedge\ v = T[w]\ \wedge\ Q[v] \neq 0)$

process $u$             { $u$ ranges over $0..N-1$ }
private var   $u.q : 0..N$
initially     $u.q = N$
do $true$ $\rightarrow$
        0: Noncritical Section;
        1: $Q[u],\ T[N-1],\ u.q := N-1,\ u,\ N-1$;
        2: do $u.q\ \neq\ 0\ \rightarrow$
        3:    await $B[u]\ \wedge\ (C(u)\ \vee\ D(u))$;
        4:    $Q[u],\ T[u.q-1],\ u.q := u.q-1,\ u,\ u.q-1$
           od;
        5: Critical Section;
        6: $Q[u],\ u.q := N,\ N$
od
```

Figure 2: Conditional mutual exclusion algorithm.

invariant $(\forall i : 0 \leq i < N :: i@\{5,6\} \ \Rightarrow\ Q[i] = 0)$

invariant $(\forall j : 0 \leq j < N :: (\mathbf{N}p :: Q[p] \leq j)\ \leq\ j+1)$

Observe that the first invariant implies that Synchrony holds, and the second and third imply that Mutual Exclusion holds. To see the latter, observe that, by substituting 0 for j in the third invariant, we have $(\mathbf{N}p :: Q[p] \leq 0) \leq 1$, which, by the second invariant, implies $(\mathbf{N}p :: p@\{5\}) \leq 1$.

Establishing the third invariant is the crux of the proof. Observe that process u may falsify this invariant only by decrementing its level, $Q[u]$, upon executing statement 4. However, as shown in the full paper, statement 3 allows process u to decrement $Q[u]$ only when $(\mathbf{N}p :: Q[p] < Q[u]) < Q[u]$ holds. This clearly implies that the third invariant is not violated. In the full paper, we give assertional proofs for the above invariants, and define a well-founded ranking to prove that the program satisfies the Progress requirement.

5 Fine-Grained Mutual Exclusion

In this section, we present a solution to the mutual exclusion problem in which processes do not busy-wait and in which only single-reader, single-writer boolean variables are used; we call such a solution *fine-grained*. Our solution to this problem is used in the proof of Lemma 3 in Section 3. As explained

```
shared var $P, Q, T$ : array$[u, v]$ of boolean
initially  $P[u] = true \;\wedge\; P[v] = true \;\wedge\; Q[u] = true \;\wedge\; Q[v] = true$
```

```
process $u$
private var $u.x$ : boolean
do $true$ $\rightarrow$
     0:   Noncritical Section;
     1:   $P[u] := false$;
     2:   $Q[u] := false$;
     3:   $u.x := T[v]$;
     4:   $T[u] := u.x$;
     5:   $P[u] := u.x$;
     6:   $Q[u] := \neg u.x$;
     7:   if $u.x$ $\rightarrow$
     8:     await $P[v]$
          [] $\neg u.x$ $\rightarrow$
     9:     await $Q[v]$
          fi;
    10:   Critical Section;
    11:   $P[u] := true$;
    12:   $Q[u] := true$
od
```

```
process $v$
private var $v.x$ : boolean
do $true$ $\rightarrow$
     0:   Noncritical Section;
     1:   $P[v] := false$;
     2:   $Q[v] := false$;
     3:   $v.x := \neg T[u]$;
     4:   $T[v] := v.x$;
     5:   $P[v] := \neg v.x$;
     6:   $Q[v] := v.x$;
     7:   if $v.x$ $\rightarrow$
     8:     await $P[u]$
          [] $\neg v.x$ $\rightarrow$
     9:     await $Q[u]$
          fi;
    10:   Critical Section;
    11:   $P[v] := true$;
    12:   $Q[v] := true$
od
```

Figure 3: Two-process mutual exclusion algorithm.

in Section 4, the mutual exclusion problem is a special case of the conditional mutual exclusion problem in which each process's enabling predicate is always identically true. For the mutual exclusion problem, the requirements given in Section 4 reduce to the following.

- *Mutual Exclusion*: $(\forall i, j : i \neq j :: CS(i) \;\Rightarrow\; \neg CS(j))$ is an invariant.
- *Progress*: $ES(i) \;\mapsto\; CS(i)$ holds for each i.

As shown in the full paper, an N-process, fine-grained solution to the mutual exclusion problem can be obtained by "nesting" $N-1$ different two-process, fine-grained solutions. The basic idea is to require each process to "compete" with each of the other $N-1$ processes in a fixed linear order. It follows that, in order to solve the N-process case, it suffices to solve the two-process case. Such a solution, consisting of two processes u and v, is depicted in Figure 3. The program is similar to the two-process solution given by Peterson in [18] and also to that given by Kessels in [12], but uses only single-reader, single-writer boolean variables.

The two variables $T[u]$ and $T[v]$ together correspond to the variable $TURN$ of Peterson's algorithm, and are used as a tie-breaker in the event that both processes attempt to enter their critical sections at the same time. Process u attempts to establish $T[u] = T[v]$ and process v attempts to establish

$T[u] \neq T[v]$. Variables $P[u]$ and $Q[u]$ are used by process u to "signal" the value of $T[u]$ to process v. $P[u]$ is used to signal that $T[u]$ is true and $Q[u]$ is used to signal that $T[u]$ is false. Observe that, while the value of $T[u]$ is being determined in statements 3 and 4 of process u, the appropriate value to signal is not known, and thus $P[u]$ and $Q[u]$ are both kept false. Also, when process u is in its noncritical section (where it may halt) $P[u]$ and $Q[u]$ are both kept true; this ensures that process v does not become forever blocked in its entry section. Variables $P[v]$ and $Q[v]$ are similarly used by process v to signal the value of $T[v]$ to process u, except their roles are reversed: $P[v]$ is used to signal that $T[v]$ is false, and $Q[v]$ is used to signal that $T[v]$ is true. The algorithm ensures that both processes never simultaneously wait on variables that are false. Avoiding such a situation is the principal problem that arises when designing a fine-grained solution to the mutual exclusion problem, as busy-waiting cannot be employed to break deadlocks.

The propositions that are needed to prove Mutual Exclusion are as follows.

$$\textbf{invariant} \quad u@\{10\} \Rightarrow (\;\; T[u] \wedge (P[v] \vee v@\{2,3\} \vee (\neg v.x \wedge v@\{4,5\})) \vee \\ \neg T[u] \wedge (Q[v] \vee v@\{3\} \vee (v.x \wedge v@\{4..6\})) \;)$$

$$\textbf{invariant} \quad v@\{10\} \Rightarrow (\;\; T[v] \wedge (P[u] \vee u@\{2,3\} \vee (u.x \wedge u@\{4,5\})) \vee \\ \neg T[v] \wedge (Q[u] \vee u@\{3\} \vee (\neg u.x \wedge u@\{4..6\})) \;)$$

From the above two invariants, we can infer that $\neg(u@\{10\} \wedge v@\{10\})$ is an invariant; this implies that the Mutual Exclusion requirement holds. In the full paper, we give assertional proofs for these invariants, and use a well-founded ranking to prove that the program satisfies the Progress requirement.

6 Necessity of Multi-Writer Variables

In this section, we establish Theorem 2 of Section 3 by showing that there exists a program that cannot be implemented by any 1-primitive program. We do so by considering a variation of the termination detection problem. In our version of this problem, there are two "worker" processes u and v and an "observer" process w. The structure of each process is shown in Figure 4. The "state" of process u is given by the shared variable UB; u is "busy" if UB is true and is "idle" otherwise. Process v's state is given by the shared variable VB, which is defined similarly.

Each of the workers u and v executes in cycles. In the beginning of each cycle, a decision is nondeterministically made to either halt, thereby leaving the given worker's state variable forever unchanged, or to continue. Note that it is possible for a worker to halt while it is in the "busy" state. The decision to continue can be made only if at least one of the workers is busy. If the given worker decides to continue, then its state variable is nondeterministically updated. This updating is preceded by an "initialization section" and followed by an "update section". These two program fragments are executed in order to inform the observer w of a possible state change. The observer executes its "waiting section" until it detects that both workers are idle, in which case it sets variable $w.done$ to true. (Note that it is possible that the two workers are never both idle.)

The conditions that must be satisfied by a program that solves this problem are as follows.

- *Reference*: Variables UB, VB, $u.done$, $v.done$, and $w.done$ cannot appear in any initialization, update, or waiting section.

```
shared var  $UB,\ VB$ : boolean
initially   $UB = true\ \wedge\ VB = true$

process $u$
private var  $u.busy,\ u.done$ : boolean
initially    $u.busy = true\ \wedge\ u.done = false$
do $true$ $\rightarrow$
    0: $u.done := \neg UB\ \wedge\ \neg VB$;
    1: if $true$        $\rightarrow$ 2: halt
       [] $\neg u.done$ $\rightarrow$ 3: skip
       fi;
    4: Initialization Section;
    5: if $true$ $\rightarrow$ 6: $UB := UB\ \vee\ VB$
       [] $true$ $\rightarrow$ 7: $UB := false$
       fi;
    8: $u.busy := UB$;
    9: Update Section
od

process $v$
private var  $v.busy,\ v.done$ : boolean
initially    $v.busy = true\ \wedge\ v.done = false$
do $true$ $\rightarrow$
    0: $v.done := \neg UB\ \wedge\ \neg VB$;
    1: if $true$        $\rightarrow$ 2: halt
       [] $\neg v.done$ $\rightarrow$ 3: skip
       fi;
    4: Initialization Section;
    5: if $true$ $\rightarrow$ 6: $VB := UB\ \vee\ VB$
       [] $true$ $\rightarrow$ 7: $VB := false$
       fi;
    8: $v.busy := VB$;
    9: Update Section
od

process $w$
private var  $w.done$ : boolean
initially    $w.done = false$
0: Waiting Section;
1: $w.done := true$
```

Figure 4: Termination detection problem.

- *Boundedness*: Each initialization, update, and waiting section must be free of unbounded **do-od** loops.
- *Termination*: Each initialization and update section is guaranteed to terminate. More formally, we require $u@\{4\} \mapsto u@\{5\}$, $u@\{9\} \mapsto u@\{0\}$, $v@\{4\} \mapsto v@\{5\}$, and $v@\{9\} \mapsto v@\{0\}$.
- *Detection*: The observer is able to "detect" that both processes are idle. More formally, define P *detects* Q to hold iff $P \Rightarrow Q$ is an invariant and $Q \mapsto P$ holds. Then, we require that $w.done$ *detects* $\neg UB\ \wedge\ \neg VB$. Observe that, by the Reference requirement and the program structure given in Figure 4, $\neg UB\ \wedge\ \neg VB$ is a stable property, i.e., once it becomes true, it remains true.

The following two lemmas are used below to prove Theorem 2.

Lemma 4: The program in Figure 5 solves the termination detection problem.

```
**shared var**  $UB,\ VB$ : **boolean**;
                $UX,\ VX$ : **boolean**
**initially**   $UB = true\ \wedge\ VB = true\ \wedge\ UX = false\ \wedge\ VX = false$

**process** $u$
**private var** $u.busy,\ u.done$ : **boolean**
**initially**   $u.busy = true\ \wedge\ u.done = false$
**do** $true\ \rightarrow$
    0: $u.done := \neg UB\ \wedge\ \neg VB$;
    1: **if** $true \rightarrow$ 2: **halt**
       $[\!]\ \neg u.done \rightarrow$ 3: **skip**
       **fi**;
    4: $UX := false$;
    5: **if** $true \rightarrow$ 6: $UB := UB\ \vee\ VB$
       $[\!]\ true \rightarrow$ 7: $UB := false$
       **fi**;
    8: $u.busy := UB$;
    9: **if** $u.busy \rightarrow$ 10: **skip**
       $[\!]\ \neg u.busy \rightarrow$ 11: $UX := true$
       **fi**
**od**

**process** $v$
**private var** $v.busy,\ v.done$ : **boolean**
**initially**   $v.busy = true\ \wedge\ v.done = false$
**do** $true\ \rightarrow$
    0: $v.done := \neg UB\ \wedge\ \neg VB$;
    1: **if** $true \rightarrow$ 2: **halt**
       $[\!]\ \neg v.done \rightarrow$ 3: **skip**
       **fi**;
    4: $VX := false$;
    5: **if** $true \rightarrow$ 6: $VB := UB\ \vee\ VB$
       $[\!]\ true \rightarrow$ 7: $VB := false$
       **fi**;
    8: $v.busy := VB$;
    9: **if** $v.busy \rightarrow$ 10: **skip**
       $[\!]\ \neg v.busy \rightarrow$ 11: $VX := true$
       **fi**
**od**

**process** $w$
**private var** $w.done$ : **boolean**
**initially**   $w.done = false$
0: **await** $UX\ \wedge\ VX$;
1: $w.done := true$
```

Figure 5: Solution to the termination detection problem.

Proof Sketch: For the program in Figure 5, the Reference, Boundedness, and Termination requirements trivially hold. In the full paper, we prove that the Detection requirement also holds by showing that $w.done\ \Rightarrow\ \neg UB\ \wedge\ \neg VB$ is an invariant and that $\neg UB\ \wedge\ \neg VB\ \mapsto\ w.done$ holds. □

Define a program to be *k-waiting* iff each of its **await** statements is either of the form "S" or "**await** B", where B is a k-writer expression. Note that the program in Figure 5 is 2-waiting since the **await** statement of process w waits on a predicate that may be modified by both processes u and v. Observe that a k-waiting program is not necessarily k-primitive. For example, the program in Figure 5 has assignments that access multiple shared variables (as required by the program structure in Figure 4), and thus is not 2-primitive.

Lemma 5: The termination detection problem cannot be solved by any 1-waiting program.

Proof Sketch: Assume, to the contrary, that there exists a 1-waiting program P that solves the termination detection problem. We derive a contradiction by showing that there exists a fair history of P in which there are infinitely many statement executions of w. This implies that the waiting section of w has an unbounded **do-od** loop, thus violating the Boundedness requirement.

The details of the proof are given in the full paper. The idea is as follows. First, we show that process u cannot become either directly or indirectly blocked on process v while $v@\{0\}$ holds. The proof is based upon the central fact that u is unable to tell whether v will decide to halt or continue. (If v decides to halt and u is blocked on v, then the Termination requirement will be violated.) By symmetry, it follows that process v cannot become either directly or indirectly blocked on process u while $u@\{0\}$ holds. Using these key facts, the required fair history can be constructed in a stepwise fashion: in each step, one of the workers is held at statement 0 and the other worker executes a complete cyle. This history is constructed so that each worker is idle infinitely often but both workers are never simultaneously idle. We show that in this history w must repeatedly check the status of each worker, i.e., w must busy-wait. □

Proof of Theorem 2: We show that the program in Figure 5 cannot be implemented by any 1-primitive program. Suppose, to the contrary, that such an implementation exists. Then, by using the initialization, update, and waiting sections of that implementation, it would be possible to construct a 1-waiting program that solves the termination detection problem. This contradicts Lemma 5. It is worth pointing out that the program in Figure 5 *can* be implemented by a 2-primitive program by using the techniques given in Lemmas 2 and 3 in Section 3. □

7 Concluding Remarks

The primary objective of this paper has been to determine how programs with **await** statements should be categorized by granularity. To this end, we presented two key results. First, we showed that any program can be implemented by a k-primitive program for some k. In a k-primitive program, each **await** statement is as simple as possible, with the exception that k-writer variables are allowed. In establishing this result, we defined and solved a new synchronization problem, the conditional mutual exclusion problem. A surprising consequence of this result is the fact that **await** statements that combine both waiting and assignment can be implemented without busy-waiting in terms of those that do not.

As a second key result, we established the existence of a program that cannot be implemented by any 1-primitive program. Together, these two results give us a means for categorizing programs by granularity: the simplest programs are those that can be "reduced" to 1-primitive ones, next are those that can be "reduced" to 2-primitive ones, etc. These results also show that for N-process programs, simple statements of the form "**await** X", "$X := y$", and "$y := X$" suffice as synchronization primitives, where y is a private boolean variable and X is a shared, single-reader, N-writer boolean variable.

Our results are not merely of theoretical interest, but also have important practical consequences. On any realistic machine, any **await** statement that has a nontrivial enabling predicate must be implemented by means of busy-waiting at some level. Our results show that the required busy-waiting is simple. Specifically, our results show that any **await** statement — no matter how complicated — can be implemented by busy-waiting on single-reader, multi-writer boolean variables (as would be required by an implementation of the statement "**await** X" of the previous paragraph). This stands in sharp contrast to the case of previous implementations, such as that given in Figure 1, where busy-waiting on complicated "global" predicates is employed. In a recent paper by Mellor-Crummey and Scott [16], it is shown that busy-waiting on global predicates is best avoided if programs are required to be scalable, as such busy-waiting induces an unacceptable degree of memory and interconnect contention.

Our results can be generalized to allow programs with **await** statements that have multiple guards. Such statements can be represented as follows.

$$\textbf{await}\ B_1 \rightarrow S_1 \ [\!] \ B_2 \rightarrow S_2 \ [\!] \ \cdots \ [\!] \ B_N \rightarrow S_N$$

This statement is atomically executed by performing some assignment S_j whose guard is true. If more than one guard is true, then the assignment to perform is selected nondeterministically. Such a statement can be implemented by using a solution to the conditional mutual exclusion problem, with "$B_1 \ \vee \ \cdots \ \vee \ B_N$" as the enabling predicate. Once inside its critical section, a process would simply select for execution an assignment whose guard is true.

In this paper, we have primarily limited our attention to determining those implementations that are possible and those that are impossible. Other issues, such as complexity and performance, are yet to be considered. In all of our implementations, statements are implemented by using mutual exclusion. This is partly due to the fact that in our main result, namely the implementation of statements of the form "**await** $B \rightarrow S$", no restrictions are placed upon the variables appearing in B or S: such a statement could conceivably reference every shared variable of a program! Without such restrictions, an implementation must ensure that only one such statement is executed at a time. By imposing restrictions on variable access, it should be possible to implement **await** statements with greater parallelism. The development of such implementations is an important avenue for further research.

Another important open question is that of precisely identifying the class of programs that can be implemented by k-primitive programs but not $(k-1)$-primitive ones. Our results merely establish that any program can be implemented by a k-primitive program for *some* k. Characterizing the class of programs that are exactly reducible to k-primitive programs would allow us to precisely categorize programs by granularity. An important special case is that of identifying the class of programs that are implementable in terms of 1-primitive programs. Because any program in this class can be implemented by a program whose statements are as fine-grained as possible, one could take membership in this class as a criterion for identifying those programs with an "acceptable" grain of interleaving. Characterizing this class of programs would thus shed light on the validity of traditional atomicity criteria such as Reynolds' Rule [3, 14, 17].

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

[3] J. Anderson and M. Gouda, "A Criterion for Atomicity", *Formal Aspects of Computing: The International Journal of Formal Methods*, to appear.

[4] 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 579, Springer Verlag, pp. 52-70.

[5] G. Andrews, *Concurrent Programming: Principles and Practice*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[6] J. Aspnes 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.

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

[8] E. Dijkstra, "A Personal Summary of the Gries-Owicki Theory", EWD554, March, 1976. In *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York, 1982.

[9] 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.

[10] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[11] A. Israeli and M. Li, "Bounded time-stamps", *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 1987, pp. 371-382.

[12] J. Kessels, "Arbitration Without Common Modifiable Variables", *Acta Informatica*, Vol. 17, 1982, pp. 135-141.

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

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

[15] 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.

[16] J. Mellor-Crummey and M. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 9, No. 1, February, 1991, pp. 21-65.

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

[18] G. Peterson, "Myths About the Mutual Exclusion Problem", *Information Processing Letters*, Vol. 12, No. 3, June, 1981, pp. 115-116.

[19] J. Peterson and A. Silberschatz, *Operating System Concepts*, Addison-Wesley, 1985.