Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization

Robert H.B. Netzer
Sanjoy Ghosh

Technical Report #1084

April 1992

In 1992 Intl Conference on Parallel Processing, August 1992, St. Charles, IL

EFFICIENT RACE CONDITION DETECTION FOR SHARED-MEMORY PROGRAMS WITH POST/WAIT SYNCHRONIZATION

Robert H.B. Netzer
Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Sanjoy Ghosh
CSRD
305 Talbot Lab
104 South Wright Street
Urbana, Illinois 61801

Abstract — *Shared-memory parallel programs are often designed to be deterministic, both in their final results and intermediate states. However, debugging such programs requires a mechanism for locating* ***race conditions*** *or violations of the intended determinacy when they occur. This paper answers a previously open question by presenting the first precise, efficient algorithm for dynamically detecting race conditions in programs that use non-trivial synchronization. We address Post/Wait synchronization, the most powerful type of synchronization for which efficient race detection is possible. Our algorithm computes the order in which synchronization operations in the execution are guaranteed to have occurred. Using this information race conditions can be detected either post-mortem or on-the-fly. Previous work has addressed either simpler types of synchronization, approximations to race detection, or a different (and easier to detect) type of race.*

INTRODUCTION

Some parallel programs are designed to be deterministic, both in their final results and intermediate states. In addition, a large class of applications naturally have deterministic implementations (such as many scientific programs). However, debugging these programs requires a mechanism for detecting violations of the intended determinacy. Points in the execution at which nondeterminacy is manifested are called *race conditions*. Race conditions occur in shared-memory programs when different processes access common memory locations in an order not fixed by the program's synchronization. We present the first efficient algorithm for precisely detecting race conditions in program executions that use non-trivial synchronization. Our algorithm uses a trace of the execution's synchronization operations to locate operations whose execution order is not fixed. Race conditions can then be detected either with an address trace (that may also be collected during execution) or by an on-the-fly analysis during a reexecution of the program.

Our primary goal is to dynamically detect race conditions that introduce any nondeterminacy. In contrast, other work has also addressed a different type of race[1]. We are considering races that occur when different processes access a common memory location in an order not guaranteed by synchronization (and at least one process modifies the location). Such races could result in either different results or paths of control in different executions on the same input. These races pinpoint those parts of the execution at which nondeterminacy is exhibited. However, some parallel programs are intended to be nondeterministic, and can exhibit a different type of race. For such programs synchronization is usually added to implement critical sections (sections of code intended to execute atomically). Without proper synchronization, shared variables accessed inside a critical section may be modified by other processes as the section executes, violating its expected atomicity. This type of interference has been called a *data race* or *access anomaly*. Since nondeterministic behavior can result, a data race is a special case of the more general race condition considered here. Making a distinction is necessary since these races have substantially different properties and require different detection techniques[9]. Henceforth, *race condition* means a race that introduces any nondeterminacy (in other work[9] we have also used the term *general race*).

Our work is novel in that it addresses Post/Wait synchronization[a], the most powerful type of non-trivial synchronization for which precise race condition detection is efficient. Previous work has addressed either conservative approximations, simpler types of synchronization, or detection of a different type of race (discussed above). Netzer and Miller proved that detecting races in executions that use synchronization powerful enough to implement mutual exclusion is NP-hard[7]. Efficient detection is possible only for weaker types of synchronization, such as Post/Wait. Pure Post/Wait synchronization is incapable of implementing mutual exclusion because Wait operations do not reset the event variable (resetting requires a Clear operation). Approximate schemes for more powerful synchronization capable of implementing mutual exclusion, such as semaphores[4] or Post/Wait with Clear operations[2], have already been proposed. Precise detection schemes for programs that use no inter-

(a) In Post/Wait synchronization, each synchronization variable (or *event variable*) is either *set* or *reset*. A *Wait(x)* blocks until x is set; a *Post(x)* sets x. Although some implementations also provide a *Clear(x)* operation to reset x, we are not considering such operations.

process synchronization (but only task spawning directives such as fork and join) have also been proposed[1, 6]. In addition, a restricted form of Post/Wait in PCF Fortran has been considered[5] (where at most one Post and Wait on any event variable are allowed). However, the full generality of Post/Wait synchronization has not yet been addressed. This paper fills the remaining gap by presenting a precise algorithm for unrestricted Post/Wait, the most powerful type of synchronization that allows efficient race detection.

Our main result is an algorithm that analyzes a trace of an execution's synchronization operations to compute its *guaranteed orderings*. The algorithm requires $O(np)$ time and space, where n is the number of synchronization operations and p is the number of processes in the execution. Intuitively, the algorithm determines that two operations have a guaranteed ordering if the semantics of Post/Wait synchronization force them to execute only in some fixed order. A race condition exists between two shared-memory accesses that reference common locations (at least one of which is modified) if they have no guaranteed ordering. No guaranteed ordering implies that the accesses might have executed in any order, introducing nondeterminacy into the execution.

We envision our algorithm being used for program debugging in one of two ways, as part of either *post-mortem* or *on-the-fly* race condition detection. First, in a pure post-mortem approach, race conditions are detected after execution ends by analyzing the execution traces. This approach requires tracing the shared-memory addresses referenced by the execution as well as its synchronization operations. From the trace of the synchronization operations, our algorithm can locate those portions of the execution that had no guaranteed ordering. From the trace of the shared-memory references, we can determine which of these portions accessed common locations and were thus race conditions. Second, in a hybrid on-the-fly/post-mortem approach, only the execution's synchronization operations are traced[3]. From this trace our algorithm can locate those sections of the execution that had no guaranteed ordering, although the shared-memory addresses referenced by them are not yet known. This information can then be used to instrument and reexecute the program to perform run-time race condition checks at these sections. Since reexecution of each process is guaranteed to be deterministic up until the first race in that process, this hybrid approach will detect at least these first races. Detecting such first races is the most important aspect of race condition detection, as subsequent races may be artifacts and not direct manifestations of bugs[8].

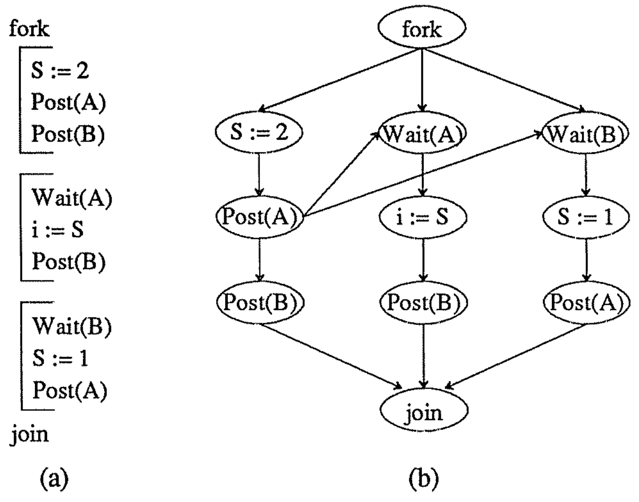
An advantage of our algorithm is that it requires only compact, efficiently obtainable traces of the execution's synchronization operations. Only the sequence of operations *per process* need be traced. No information about the relative execution order of operations in different processes is required. Execution traces are kept compact because the required instrumentation does not maintain or trace the value of any clock. Each synchronization operation requires tracing only a 1-bit flag indicating the operation type (Post or Wait) and the 4-byte event variable address. Preliminary experiments show that execution-time tracing overhead is typically less than 10%, and because the traces are compact, long executions can be traced.

Although our emphasis is on race detection, our algorithm can also provide information necessary for checking assertions about the execution order of operations in different processes. For example, a programmer may wish to check if some access must always occur after another. Our algorithm computes such information.

EXAMPLE

As motivation, we present an example program below and analyze its execution to illustrate guaranteed orderings. We use the *guaranteed ordering graph* to represent these orderings. In following sections we formally define the guaranteed orderings, and present our algorithm for computing the guaranteed ordering graph.

Consider the program in Figure 1(a), which spawns three parallel processes, among which the variable "S" is shared. Although these processes execute in parallel, their Post and Wait operations force some operations to execute before others. The graph in Figure 1(b) (discussed below) represents these orderings. Consider an execution in which the Post(A) in the last process executes before the Wait(A), causing "S := 1" to execute before "i := S". A naive analysis of this execution might


```
fork
  S := 2
  Post(A)
  Post(B)

  Wait(A)
  i := S
  Post(B)

  Wait(B)
  S := 1
  Post(A)
join
```

Figure 1. (a) program, (b) guaranteed ordering graph

suggest that no race condition exists between these references to S, because synchronization caused "S := 1" to precede "i := S". However, this particular ordering is not *guaranteed* to occur, because this Post(A) *can* execute after the Wait(A). We can determine this by noticing that all operations in the first process (which includes another Post(A)) could execute first, allowing all operations in the other processes to proceed. Thus, there is a race condition between "S := 1" and "i := S' because they could execute in either order, introducing nondeterminacy into the execution. However, no such race exists between "S := 2" and "i := S". Unlike the Post(A) in the last process, the Post(A) in the first process is *guaranteed* to precede the Wait(A). We can determine this by noticing that neither the Wait(A) nor Wait(B) operations can proceed until after the Post(A) is issued by the first process. The assignment "S := 2" is thus guaranteed to always precede "i := S" and "S := 1"; no race involving "S := 2" exists.

Analyzing each synchronization operation in the execution in this way is the problem of computing guaranteed orderings that we are addressing. The algorithm presented later computes these orderings by considering each operation, *a*, and locating the set of operations that could execute before *a*. Then, any operation not in this set is guaranteed to execute after *a*.

We will represent an execution's guaranteed orderings with the *guaranteed ordering graph*, as illustrated in Figure 1(b). Each node represents the execution instance of a synchronization operation or one or more (consecutively executed) program statements. Edges represent the guaranteed orderings. Edges always exist from each node to the next node in the same process. An edge also exists between nodes in different processes if one is guaranteed to precede the other. The graph has a path from node *a* to another node *b* iff *a* is guaranteed to precede *b*. For example, the edge in Figure 1(b) from Post(A) in the first process to Wait(A) and Wait(B) indicates that Post(A) is guaranteed to precede Wait(A) and Wait(B). In addition, the absence of a path between "i := S" and "S := 1" indicates that no guaranteed ordering between them exists.

PROBLEM STATEMENT

Given a trace of the execution's synchronization operations, the algorithm presented later determines which operations do not have guaranteed orderings. Below we formally characterize what it means for the ordering between operations to be guaranteed. We define a *must-have-happened-before* relation, $\xrightarrow{\text{MHB}}$, on the execution's synchronization operations. Intuitively, an operation *a* must happen before another operation *b* iff the semantics of Post/Wait synchronization could not allow *b* to complete before *a* begins execution.

We base the guaranteed orderings on the execution's *explicit* synchronization, and assume that shared memory is not used to implement implicit synchronization. For programs that contain implicit synchronization, we might omit some orderings that are in fact guaranteed. In addition, Netzer and Miller have shown that *unintentional* implicit synchronization can occur when shared values are used in conditional expressions[8]. They present techniques to estimate additional orderings introduced by intentional or unintentional implicit synchronization. These techniques are complementary to our algorithm, so we will not address this issue here.

Consider the synchronization operations performed by the single program execution being analyzed. We formalize how the semantics of these operations might allow various orderings to occur. Different orderings can occur because of variations in the execution speed of each process. Intuitively, the guaranteed orderings are those that always occur regardless of these variations. Two factors influence orderings:

(1) The i^{th} operation in process p (denoted $e_{p,i}$) cannot execute until after the $i-1^{\text{st}}$ operation ($e_{p,i-1}$).

(2) The execution order of operations belonging to different processes is influenced by Post and Wait operations: a Wait(x) cannot complete until after the execution of a Post(x) has begun.

We capture these factors by defining for a synchronization operation S the *maximal* set of other operations in the execution that could possibly execute before S. Any operation that satisfies these two points (without requiring S to first begin execution) *could* complete before S. Any operation that does not meet these requirements must wait until after S begins before proceeding. We capture this latter set of operations by defining a *maximal valid execution* for S, denoted $MVE(S)$:

Definition 1

$MVE(S)$ is a maximal sequence of operations in the execution that the semantics of Post/Wait synchronization *could* allow to execute before S, and contains every operation $e_{p,i}$ such that

(1) $e_{p,i}$ is not S, and
(2) all events preceding $e_{p,i}$ in process p also precede $e_{p,i}$ in $MVE(S)$, and
(3) $e_{p,i}$ is either a Post, or is a Wait(x) and a Post(x) precedes it in $MVE(S)$. ■

We could compute an $MVE(S)$ by mimicing the way an actual execution would advance. The next operation not yet executed in each process is eligible to proceed; Post operations can always proceed (and can thus be added to $MVE(S)$), but Wait(x) operations must block until a Post(x) has been issued. Since $MVE(S)$ is *maximal*, it contains as many operations as possible subject to these constraints. Although the order in which operations can be added to $MVE(S)$ is not unique (since we can pick the next operation in any process to consider), all $MVE(S)$ contain the same, unique set of operations. Our algorithm (presented below) computes this set to determine the guaranteed orderings.

We can now define those orderings that are guaranteed to occur regardless of timing variations. $MVE(S)$ contains exactly those synchronization operations that *could* execute before S, but are not guaranteed to do so. Thus, operations not in $MVE(S)$ are exactly those that are *guaranteed* to follow S. We define the *must-have-happened-before* relation, $\xrightarrow{\text{MHB}}$, on the synchronization operations to represent these orderings.

Definition 2

For any two synchronization operations a and b in the execution,

$$a \xrightarrow{\text{MHB}} b \quad \Leftrightarrow \quad b \notin MVE(a). \blacksquare$$

ALGORITHM

We now present our algorithm and show that it requires $O(np)$ time and space (where n is the total number of operations and p is the number of processes in the execution). For simplicity, we assume that only Post and Wait operations are issued by the execution. The algorithm can easily be extended to handle task spawning and destruction operations (such as fork and join). A correctness argument for the algorithm appears in the Appendix.

Algorithm 1 analyzes a trace of the execution's synchronization operations to compute the $\xrightarrow{\text{MHB}}$ relation. The traces must contain the sequence of operations performed by each process, and indicate the operation type (Post or Wait) and the event variable. As discussed earlier, because the relative order of operations in different processes is not required, these traces are compact and can be efficiently collected even for long executions.

Algorithm 1 constructs the guaranteed ordering graph (illustrated earlier). Recall that this graph contains one node for each operation performed during execution, and edges to represent the guaranteed orderings. By definition, an edge always exists from each operation to the next operation in the same process (these edges are not added by the algorithm). The algorithm adds edges between different processes to show the guaranteed orderings. The resulting graph has a path from an operation a to another operation b iff $a \xrightarrow{\text{MHB}} b$. This graph is a convenient representation of $\xrightarrow{\text{MHB}}$ as it requires only $O(np)$ space (each of the n nodes has at most p out-edges), and it allows *vector timestamps* to be computed that allow constant-time determination of whether any two operations have a guaranteed ordering[4].

Algorithm 1 computes this graph by considering every operation in the execution and *visiting* the operations in its *MVE*. Visiting operations consists of simulating their execution by honoring the semantics of Post/Wait synchronization. After visiting the *MVE* for an operation $e_{p,i}$, an edge is added from $e_{p,i}$ to the first operation not in $MVE(e_{p,i})$ in each process. Such edges establish paths from $e_{p,i}$ to every operation not in $MVE(e_{p,i})$, which exactly captures the definition of $\xrightarrow{\text{MHB}}$.

```
1:  ComputeGuaranteedOrderings:
2:      First = the set containing the first op in each process;
3:      for p = 1 to the number of processes {
4:          Visitable = the set of Post ops in First;
5:          for each event variable, x
6:              Untrig(x) = the set of Wait(x) ops in First;
7:              IsTrig(x) = FALSE;
8:          for i = 1 to the number of ops in process p {
9:              VisitMVE($e_{p,i}$);
10:             add an edge in the graph from $e_{p,i}$ to every op
11:                 in each Untrig set;
12:         }
13:     }

14: VisitMVE($e_{p,i}$):
15:     while ( Visitable contains ops other than $e_{p,i}$ ) {
16:         remove any op, e, from Visitable (except $e_{p,i}$);
17:         if e is a Post(x) then
18:             move all ops in Untrig(x) to Visitable;
19:             IsTrig(x) = TRUE;
20:         if an op, e', follows e in the same process
21:             if e' is a Wait(x) and IsTrig(x) = FALSE
22:             then
23:                 add e' to Untrig(x);
24:             else
25:                 add e' to Visitable;
26:     }
```

Algorithm 1. Compute guaranteed ordering graph (''op'' means ''operation'')

During the p^{th} iteration of the outer **for** loop, ''VisitMVE'' (line 14) is iteratively called to visit the *MVE*'s for all operations in process p. In VisitMVE, operations belonging to the same process are visited in intra-process order, and a Wait(x) operation is not visited until it is *triggered* by a Post(x) operation. The set *Visitable* contains operations eligible for visitation. An operation is visited after it is removed from this set (line 16). The set $Untrig(x)$ contains Wait(x) operations that are ready to be visited but are not yet triggered. The Boolean array *IsTrig* records which event variables have been triggered. When a Post(x) is visited, all untriggered Wait(x) operations are moved from $Untrig(x)$ to *Visitable*.

VisitMVE visits as many operations as possible without visiting $e_{p,i}$, the operation passed as the parameter. To visit the *MVE* for the first operation in some process p (i.e., $MVE(e_{p,1})$), the sets *Visitable*, *Untrig*, and *IsTrig* are initialized (lines 4–7) to reflect the initial state of the execution's synchronization: the first Post operation in each process is eligible for visitation, and all Waits are untriggered (because event variables are initially reset). To visit the *MVE* of subsequent operations in process p (i.e., $MVE(e_{p,i})$ for $i > 1$), we notice that $MVE(e_{p,i})$ is a superset of $MVE(e_{p,i-1})$ so it is unnecessary to start the visit from scratch. After VisitMVE is called with $e_{p,i}$, it will return when *Visitable* contains only $e_{p,i}$, indicating that as many operations as possible have been visited without visiting $e_{p,i}$ (which is exactly $MVE(e_{p,i})$). Then, when VisitMVE is called with $e_{p,i+1}$, $e_{p,i}$ will finally be visited, possibly causing more operations to be added to *Visitable*, until finally $MVE(e_{p,i+1})$ has been visited.

To analyze the algorithm's running time, assume that the sets *Visitable* and *Untrig* are implemented as linked lists, allowing constant-time set operations, such as checking the emptiness of *Visitable* (line 15), adding or removing operations (lines 16, 23, 25), and moving all operations in $Untrig(x)$ to *Visitable* (line 18). Each iteration of the **while** loop thus requires constant time. Next consider a single iteration of the outer **for** loop (line 3). The first inner **for** loop (line 5) performs $O(e)$ work (where e is the number of event variables) to initialize the sets. The second inner **for** loop (line 8) performs $O(n)$ work by calling VisitMVE for each operation in process p, causing every operation in the execution to be visited once (except the last operation in p, which is not visited). Each iteration of the outer **for** loop thus performs $O(n+e) = O(n)$ work. Since the outer **for** loop iterates p times, $O(np)$ total work is performed. As mentioned above, the graph requires $O(np)$ space.

CONCLUSION

Our algorithm pinpoints parts of the execution at which nondeterminacy is introduced. Identifying these points is essential for debugging programs that are intended to be deterministic. The algorithm is an essential component of either a post-mortem or hybrid on-the-fly/post-mortem race detection tool for programs with Post/Wait synchronization. The algorithm also provides information useful for checking assertions about the order in which operations are expected to execute. In addition, the core of the algorithm, which visits the *MVE* for an operation, can be used to derive other algorithms. For example, we can determine in $O(n)$ time and constant space whether a guaranteed ordering exists between two given operations. We can also locate all pairs of operations that have no guaranteed ordering in $O(np^2)$ time and $O(p)$ space. Moreover, when applied to more powerful synchronization (such as semaphores), our algorithm conservatively computes the guaranteed orderings.

ACKNOWLEDGEMENTS

Robert Netzer was supported in part by NSF grants CCR-8815928 and CCR-9100968, ONR grant N00014-89-J-1222, and by Sequent Computer Systems. Sanjoy Ghosh was supported by grant AFOSR 90-0044 FY92 from the Air Force Office of Scientific Research.

REFERENCES

[1] Dinning, A. and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *2nd ACM Symp. on Princ. and Practice of Parallel Prog.*, pp. 1-10 Seattle, WA, (March 1990).

[2] Emrath, P.A., S. Ghosh, and D.A. Padua, "Event Synchronization Analysis for Debugging Parallel Programs," *Supercomputing '89*, pp. 580-588 Reno, NV, (November 1989).

[3] Emrath, P.A., S. Ghosh, and D.A. Padua, "On-the-fly Race Detection for Parallel Programs with Events," *1992 Intl. Conf. on Parallel Processing*, St. Charles, IL, (August 1992).

[4] Helmbold, D.P., C.E. McDowell, and J.-Z. Wang, "Analyzing Traces with Anonymous Synchronization," *1990 Intl. Conf. on Parallel Processing*, pp. 70-77 St. Charles, IL, (August 1990).

[5] Hood, R., K. Kennedy, and J. Mellor-Crummey, "Parallel Program Debugging with On-the-fly Anomaly Detection," *Supercomputing '90*, pp. 74-81 New York, NY, (November 1990).

[6] Mellor-Crummey, J.M., "On-the-Fly Detection of Data Races for Programs with Nested Fork-Join Parallelism," *Supercomputing '91*, pp. 24-33 Albuquerque, NM, (November 1991).

[7] Netzer, R.H.B. and B.P. Miller, "On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions," *1990 Intl. Conf. on Parallel Processing*, pp. II-93–II-97 St. Charles, IL, (August 1990).

[8] Netzer, R.H.B. and B.P. Miller, "Improving the Accuracy of Data Race Detection," *3rd ACM Symp. on Princ. and Practice of Parallel Prog.*, pp. 133-144 Williamsburg, VA, (April 1991).

[9] Netzer, R.H.B. and B.P. Miller, "What are Race Conditions? Some Issues and Formalizations," *ACM Letters on Programming Languages and Systems* 1(1)(March 1992).

APPENDIX

In this appendix we argue that our algorithm is correct. We first make the following observations about the algorithm. For brevity, we will let $Visited(e_{p,i})$ denote the sequence of operations that have been visited at the end of the i^{th} iteration of the inner **for** loop (line 10), during the p^{th} iteration of the outer **for** loop.

Observation 1. If a node s is in $Visited(e_{p,i})$, then all preceding operations in the same process also precede it in $Visited(e_{p,i})$. This follows since if s is not the first operation in its process, s is added to *Visitable* (line 23) only after the preceding operation is removed (line 15).

Observation 2. If a Wait(x) operation is in $Visited(e_{p,i})$, then a Post(x) precedes it in $Visited(e_{p,i})$. This follows since a Wait(x) is added to *Visitable* (line 23) only after $IsTriggered(x)$ becomes TRUE, which occurs only after a Post(x) is removed from *Visitable* (lines 16–18).

Observation 3. If s is a Post(x) and all preceding operations in the same process precede it in $Visited(e_{p,i})$, then s is also in $Visited(e_{p,i})$ (or $s = e_{p,i}$). This follows since Post operations are always added to *Visitable* (line 23).

Observation 4. If s is a Wait(x), and all operations preceding it in the same process, and a Post(x), precede it in $Visited(e_{p,i})$, then s is also in $Visited(e_{p,i})$ (or $s = e_{p,i}$). This follows since Wait(x) operations are added to *Visitable* if $IsTriggered(x)$ is TRUE (line 21), and the prior visit of the Post(x) would have set $IsTriggered(x)$ to TRUE.

Lemma 1

For all operations $e_{p,i}$, $Visited(e_{p,i})$ always contains the same unique set of operations as any $MVE(e_{p,i})$.

Proof.

We prove that any $MVE(e_{p,i})$ contains an operation iff $Visited(e_{p,i})$ always contains the same operation. The uniqueness of the set of operations they contain then follows.

If Part. We must show that every operation in $Visited(e_{p,i})$ is always in any $MVE(e_{p,i})$. Consider any $MVE(e_{p,i})$, denoted M. M and $Visited(e_{p,i})$ will share a (possibly empty) common prefix. We will show that each operation in the suffix of $Visited(e_{p,i})$ exists somewhere in M, by induction on the position, n, in the suffix.

Basis (n=1). We are considering the first element, s, in the suffix of $Visited(e_{p,i})$ that differs from M. If s is a Post(x) operation, then all preceding operations in the same process also precede s in $Visited(e_{p,i})$ (Observation 1) and are thus in M. By part (2) of the definition of *MVE*, s must also be in M. If s is a Wait(x) operation, all preceding operations in the same process precede s in $Visited(e_{p,i})$ (Observation 1), and a Post(x) precedes s in $Visited(e_{p,i})$ (Observation 2). These operations are thus also in M, so s must be in M (by parts (2) and (3) of the definition of *MVE*).

Induction. Assume that the first n operations in the suffix of $Visited(e_{p,i})$ exist somewhere in M. We must show that the $n+1^{th}$ operation, s, is also in M. As above, if s is a Post(x) operation, then all preceding operations in the same process precede s in $Visited(e_{p,i})$. By the induction hypothesis, these operations are in M, and by part (2) of the definition of *MVE*, s must also be in M. If s is a Wait(x) operation, all preceding operations in the same process, and a Post(x), precede s in $Visited(e_{p,i})$. By the induction hypothesis, these operations are in M, and by parts (2) and (3) of the definition of *MVE*, s is also in M.

Only if part. We must show that each operation in any $MVE(e_{p,i})$ always belongs to $Visited(e_{p,i})$. As above, consider any $MVE(e_{p,i})$, denoted M. M and $Visited(e_{p,i})$ will share a (possibly empty) common prefix. We will show that each operation in the suffix of M exists somewhere in $Visited(e_{p,i})$, by induction on the position, n, in the suffix.

Basis (n=1). Consider the first element, s, in the suffix of M that differs from $Visited(e_{p,i})$. If s is a Post(x) operation, then all preceding operations in the same process also precede s in M (part (2) of the definition of *MVE*) and are thus in $Visited(e_{p,i})$. By Observation 3, s must also be in $Visited(e_{p,i})$. If s is a Wait(x) operation, all preceding operations in the same process precede s in M (part (2) of the definition of *MVE*), and a Post(x) also precedes s in M (part (3) of the definition of *MVE*). These operations are also in $Visited(e_{p,i})$, and by Observation 4, s must also be in $Visited(e_{p,i})$.

Induction. Assume that the first n operations in the suffix of M exist somewhere in $Visited(e_{p,i})$. We must show that the $n+1^{th}$ operation, s, is also in $Visited(e_{p,i})$. As above, if s is a Post(x) operation, all preceding operations in the same process precede s in M, and by the induction hypothesis these operations are in $Visited(e_{p,i})$. By Observation 3, s must also be in M. If s is a Wait(x) operation, all preceding operations in the same process, and a Post(x), precede s in M. By the induction hypothesis, these operations are in $Visited(e_{p,i})$, and by Observation 4, s must also be in $Visited(e_{p,i})$. ■

Lemma 2

For all operations $e_{p,i}$ and $e_{q,j}$, if Algorithm 1 adds an edge from $e_{p,i}$ to $e_{q,j}$, then $e_{q,j} \notin MVE(e_{p,i})$.

Proof.

From Lemma 1 we know that during the p^{th} iteration of the outer **for** loop, after the i^{th} iteration of the inner **for** loop, exactly $MVE(e_{p,i})$ has been visited. Since Algorithm 1 adds (in line 10) an edge to the first operation *not* visited in each process, every edge from $e_{p,i}$ is to an operation $e_{q,j} \notin MVE(e_{p,i})$. ■

Lemma 3

If $a \notin MVE(b)$ and $b \notin MVE(c)$, then $a \notin MVE(c)$.

Proof.

For a contradiction, assume that $a \in MVE(c)$. Consider the last operation in each process that belongs to $MVE(c)$. Since the subsequent operations are not in $MVE(c)$, they are not needed to trigger a (which we know since $a \in MVE(c)$). Let b' be the last such operation in b's process ($b' \in MVE(c)$ but $b \notin MVE(c)$). Since no operation following b' in this process is needed to trigger a, $a \in MVE(b')$. Because b follows b' in the same process, we must also have $a \in MVE(b)$, contradicting the assumption. Thus, we must have $a \notin MVE(c)$. ■

Theorem 1.

Algorithm 1 is correct. That is, for all operations $e_{p,i}$ and $e_{q,j}$, a path exists from $e_{p,i}$ to $e_{q,j}$ after the algorithm terminates iff $e_{p,i} \xrightarrow{\text{MHB}} e_{q,j}$ (or $e_{q,j} \notin MVE(e_{p,i})$).

Proof.

If part. We must show that for all $e_{p,i}$ and $e_{q,j}$, if $e_{q,j} \notin MVE(e_{p,i})$, then a path exists from $e_{p,i}$ to $e_{q,j}$. From Lemma 1 we know that during the p^{th} iteration of the outer **for** loop, after the i^{th} iteration of the inner **for** loop, exactly $MVE(e_{p,i})$ has been visited. Algorithm 1 adds an edge to the first operation *not* visited in each process, so every edge from $e_{p,i}$ is to an operation $e_{q,j} \notin MVE(e_{p,i})$. Because there is an edge from each operation to the subsequent operation in the same process, there will be a path from $e_{p,i}$ to *all* events not in $MVE(e_{p,i})$. Since $e_{q,j} \notin MVE(e_{p,i})$, a path will exist from $e_{p,i}$ to $e_{q,j}$.

Only If part. We must show that for all $e_{p,i}$ and $e_{q,j}$, if a path exists from $e_{p,i}$ to $e_{q,j}$ then $e_{q,j} \notin MVE(e_{p,i})$. We use induction on the length of the path.

Basis (path of length 1). Follows directly from Lemma 2.

Induction: Assume that the hypothesis holds for paths of length n. We must prove that if a path of length $n+1$ exists from $e_{p,i}$ to $e_{r,k}$ (because of a path of length n from $e_{p,i}$ to $e_{q,j}$ and an edge from $e_{q,j}$ to $e_{r,k}$), then $e_{r,k} \notin MVE(e_{p,i})$. The path of length n from $e_{p,i}$ to $e_{q,j}$ implies that $e_{q,j} \notin MVE(e_{p,i})$ (by the induction hypothesis). The edge from $e_{q,j}$ to $e_{r,k}$ implies that $e_{r,k} \notin MVE(e_{q,j})$ (by Lemma 2). Since $e_{r,k} \notin MVE(e_{q,j})$ and $e_{q,j} \notin MVE(e_{p,i})$, by Lemma 3 we have $e_{r,k} \notin MVE(e_{p,i})$. ■