A Bounded First-In, First-Enabled Solution to the ℓ-Exclusion Problem

Yehuda Afek
AT&T Bell Laboratories and Tel-Aviv University
and
Danny Dolev
IBM Almaden Research Center and Hebrew University
and
Eli Gafni
University of California, Los Angeles
and
Michael Merritt
AT&T Bell Laboratories
and
Nir Shavit
Tel-Aviv University

This paper presents a solution to the *first-come, first-enabled* ℓ-exclusion problem of [?]. Unlike the solution in [?], this solution does not use powerful read-modify-write synchronization primitives, and requires only bounded shared memory. Use of the *concurrent timestamp system* of [?] is key in solving the problem within bounded shared memory.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Mutual exclusion*

Additional Key Words and Phrases: concurrency, time stamps, atomic registers.

1. INTRODUCTION

Consider a system of n asynchronous processes that communicate only by reading from and writing to bounded atomic registers. The program of each process contains a piece of code called the *critical section.* The ℓ-exclusion problem is to guarantee that the system does not enter a global state in which more than ℓ processes are executing their critical section [?].

To illustrate the problem, imagine that each process controls some device which from time to time needs to enter a mode of high electrical power consumption. The main circuit breaker can withstand at most ℓ devices at high electrical power consumption. By allowing each process to switch its device on only when it is in its

Eli Gafni was supported by NSF Presidential Young Investigator Award under grant DCR84-51396 & matching funds from XEROX Co. under grant W881111. This work was performed while this author was at Tel-Aviv University. Parts of this research were conducted while Nir Shavit was at the Hebrew Univerity, IBM Almaden Research Center and Stanford University, and visiting AT&T Bell Laboratories and MIT.

critical section, an ℓ-exclusion solution will protect the circuit breaker from burning out.

The ℓ-exclusion problem is an extension of mutual exclusion (where $\ell = 1$), a classic problem in concurrency control [?, ?]. This extension was first defined and solved by Fischer, Lynch, Burns, and Borodin in [?]. A solution is required to withstand the slow-down or even the crash of processes (up to $\ell - 1$ of them), and should not require the active collaboration of processes that are not currently requesting a resource. (Solutions to the more difficult *ℓ-assignment* problem [?] must assign each process in the critical section to a distinct "slot", in addition to maintaining the ℓ-exclusion property.)

Lamport observed [?] that a first-come, first-served mutual exclusion algorithm can be constructed by preceding a mutual exclusion mechanism with a fifo-queue. The possibility of faults precludes the straightforward use of any such first-come, first-served mechanism–the first process to enter the mechanism may fail, preventing progress by those following [?, ?], To resolve this dilemma, the notion of process enabling was introduced: a process is *enabled* to enter the critical section if sufficiently many steps of that process will carry it into the critical section, independently of the actions of other processes. "The key distinction between enabling and actual entry to the critical section is that a process might become enabled passively by the action of some other process changing shared memory, whereas entry to the critical section can take place only by a positive action of the given process," [?, ?]. The first-in, first-enabled condition was introduced as the natural fairness condition for ℓ-exclusion, generalizing the first-come, first-served condition for mutual-exclusion.

The difficulty of solving fault-tolerant distributed problems such as ℓ-exclusion depends critically on the available concurrent objects from which solutions are to be constructed. A recent paper by Herlihy [?] established the existence of a hierarchy of concurrent objects. Herlihy's hierarchy classifies objects according to the number of processes among which these objects can solve the consensus problem, despite any number of faults. An object has a consensus number k if k is the maximum number of asynchronous fail-stop processes for which the object can be used to solve the consensus problem. Thus objects with higher consensus number cannot be deterministically and fault-tolerantly implemented by employing objects with lower consensus numbers. The consensus number of atomic read/write shared memory is one.

The only formerly known first-in, first-enabled solution to the ℓ-exclusion problem, due to [?, ?], was defined and solved based on the use of *read-modify-write* primitives, whose consensus number is ∞. Rudolph, in [?], solved the problem assuming the weaker *fetch-and-add* primitive, whose consensus number is two. In [?, ?, ?], bounded solutions were presented using only atomic shared memory, but they implement a weaker form of fairness, *n^2-overtaking*. (A process may be overtaken by as many as n^2 later processes, before becoming enabled. First-in, first-enabled corresponds to 0-overtaking. These notions are formally defined in Section ??.)[1]

Though bounded-size memory *first-come, first-served* solutions to the mutual exclusion problem were presented in [?, ?], it was not known how to generalize these techniques to the case of ℓ-exclusion. It seems that in order to achieve first-in, first-

[1]An appendix of [?] describes an earlier version of the algorithm of this paper, and erroneously claims it to be first-in, first-enabled.

enabled fairness, a process must be able to deduce the relative order among other processes, not only between itself and others, and this must be done in the face of concurrent state changes by the other processes. It has remained open whether any shared-memory solution having less than n^2-overtaking was achievable, without resorting to the use of communication primitives of greater power than atomic read/write shared memory. (Henceforth, atomic read/write shared memory will be referred to as *shared memory.*)

This paper presents a first-in, first-enabled solution to the ℓ-exclusion problem, using only shared memory as communication primitives. The definitions and proofs are all natural generalizations of mutual exclusion. The paper contains a complete solution that uses unbounded (integer) values in the shared memory as timestamps (similar to the ticket algorithm for mutual-exclusion [?]). In addition, we indicate how these unbounded values can be replaced by implementations of *concurrent timestamp systems* [?] from bounded shared memory, such as those presented in [?, ?, ?, ?]. (In these cases, the resulting solution requires $O(n)$ bits of shared memory per process.)

2. THE ℓ-EXCLUSION PROBLEM

A *system* consists of n processes $\{1, \ldots, n\}$. Each *process* is a state machine, possibly with an infinite number of states. We consider an interleaving model of concurrency, where executions are modeled as sequences of *steps.* Each step is performed by a single process. A process i performs either a write step, $x := v$, or a read step, $v := x$, where v is a local variable and x is a shared register, or performs some local computation, and moves to a new state, from which one or more subsequent steps are enabled. *Process states* consist of the values of local variables and of the control (program counter). A *system state* consists of the states of the processes and registers. For convenience, we assume that each process can take at least one step from any system state.

A *run* is an alternating sequence (finite or infinite) of states and process steps, $s_0, e_1, s_1, \ldots$, where each event (the triple s_{i-1}, e_i, s_i) corresponds to a step of a single process in the obvious way.

We assume that each process can be described by a program that consists of two distinguished sections: a *remainder section* and a *critical section.* Each process alternates between executing its remainder and its critical section as follows:

```
Process $i$:
repeat forever
    remainder_section$_i$
    critical_section$_i$
end repeat
```

It is convenient to assume that these sections begin and end with local steps of i, e.g. `Begin_remainder_section`$_i$ and `End_remainder_section`$_i$. If i has taken a `Begin_remainder_section`$_i$ step, we say i has *entered* the remainder section. If i has taken a `End_remainder_section`$_i$ step, we say i has *left* the remainder section, and in between i is *in* the remainder section. Similar notation is used later for other sections of code. Two or more processes are *concurrently in* particular sections of code in a system state if they are all in the given sections in the given state.

If a process i takes only finitely many steps in a given run of the system and stops outside the remainder section, or enters the critical section and never leaves, we say i is *faulty* in the run.

The ℓ-exclusion problem is to guarantee that the system does not enter a global state in which more than ℓ processes are in their critical section [?]. To coordinate the entrance to the critical section, *entry* and *exit* sections are added to the program of each process:

```
Process i:
repeat forever
        remainder_section_i
        entry_i
        critical_section_i
        exit_i
end repeat
```

The following properties are required from any solution to the problem:

DEFINITION 2.1. ℓ-Exclusion: *No more than ℓ processes are ever concurrently in their critical sections.*

DEFINITION 2.2. ℓ-Lockout Avoidance: *If fewer than ℓ processes are faulty, any process that is not faulty and leaves the remainder region later re-enters it.*

(Note that if ℓ or more processes are faulty and stay in the critical section forever, the ℓ-exclusion condition requires that no other process enter its critical section. Hence, the definition of ℓ-lockout avoidance only requires progress when less than ℓ processes are faulty.)

Lockout avoidance can be strengthened in the following way. The definition of the entry section is refined to consist of two sections, a *doorway* and a *waiting room*. Any process must pass through the doorway in a bounded number of steps, whether or not other processes take steps. Execution of the waiting room may consist of an unbounded number of steps, as a process is busy waiting there for space in the critical section to become available.

```
Process i:
repeat forever
        remainder_section_i
        doorway_i
        waiting_room_i
        entry_i
        critical_section_i
        exit_i
end repeat
```

The strengthened fairness property imposes the following constraint:

DEFINITION 2.3. First-Come, First-Served: *If process i leaves the doorway before process j enters the doorway, then i enters the critical section before j.*

As noted by [?], the first-come, first-served property and the ℓ-lockout avoidance property cannot be mutually satisfied when ℓ is greater than one. To see this, suppose process i is required by the first-come, first-served property to enter its critical section before process j, but process i fails. Since process failures are not detectable, process j must wait forever, even if there are no other processes in their critical sections. This violates the ℓ-lockout avoidance property when $l > 1$. Thus, for the general ℓ-exclusion problem, the first-come, first-served condition must be weakened. Rather than requiring the earliest process to be serviced first, we require instead that it be *enabled* first.

Definition 2.4. *Process i is* enabled *(to enter the critical section) in a system state s, if in any run that starts in s and in which i takes an infinite number of steps (regardless of the number of faults), i enters the critical section.*

Note that an enabled process remains enabled until it enters its critical section. Thus, lockout avoidance can be strengthened to the following property:

Definition 2.5. First-In, First-Enabled: *If i last left the doorway before j last entered it, i is in the waiting room, and j is in the critical section, then i is enabled.*

Note that for $\ell = 1$, where ℓ-exclusion reduces to mutual exclusion, the first-in, first-enabled property implies the first-come, first-served property.

The algorithm of Section ?? actually satisfies a stronger, *bounded* first-in, first-enabled condition: for any finite run α, if i last left the doorway before j last entered it, i is in the waiting room, and j is in the critical section, then there exists $k \geq 0$ such that in any extension of α in which i takes k additional steps, and regardless of the number of faults, i enters its critical section.

The first-in, first-enabled fairness condition can be weakened by allowing some bounded number of processes to pass a process in the waiting room:

Definition 2.6. *Process i is* overtaken *by process j in a finite run if i last left the doorway before j last entered it, i is in the waiting room, j is in the critical section, and i is not enabled.*

An ℓ-exclusion algorithm is *k-overtaking* if there is a run in which a process is overtaken by other processes k distinct times in a single pass through the waiting room. As we observe above, the bounded solutions of [?, ?, ?] are n^2-overtaking.

In these algorithms, there could be $\ell - 1$ processes in the critical section, a process i may leave the doorway, and if run alone, enter the critical section. But while i is still in the waiting room, another process could pass through the doorway and enter the critical section. Then i cannot enter the critical section, no matter how many steps it takes, until at least one of the other processes leaves the critical section. Despite the *possibility* of entering the critical section once it left the doorway, the *possibility* of being blocked means i is not yet enabled.

3. THE SOLUTION

The algorithm presented in Figure 2 is a first-in, first-enabled solution to the ℓ-exclusion problem. It uses the procedures of Figure 1, which implement a *concurrent timestamp system.*

3.1 Concurrent Timestamp Systems

A concurrent timestamp system provides some of the semantics of a fetch-and-increment abstraction, such as might be used in ticket algorithms [?, ?, ?]. Fetch-and-increment can be used to assign unique integer values (timestamps) as each process passes through the doorway. The timestamps can then be used to assign priorities to processes for entry to the critical section. Unfortunately, fetch-and-increment assumes registers of unbounded size. Even if it were defined using modular arithmetic, it cannot be implemented from shared memory without introducing waiting, and so is not fault-tolerant [?].

As we show, the unbounded values and strong atomic semantics of fetch-and-add are not necessary in the context of ℓ-exclusion. Concurrent timestamp systems are an abstract mechanism for resolving the relative priority of competing processes, without the drawbacks of fetch-and-add.

A concurrent timestamp system is an object shared by the processes, each of which interacts with it via two procedures, `Label` and `Scan`. The `Label` procedures return no values, and the `Scan` procedures return a permutation of the n process identifiers. Concurrent timestamp systems are defined formally below. Intuitively, in the $\texttt{Label}_i$ procedure, a process i updates a stored timestamp to be later than all the timestamps previously reserved by other processes. Since the $\texttt{Label}_i$ procedure does not return a value, this timestamp is not directly visible to the calling procedure. The timestamp system stores the array of timestamps, ordered according to a serialization of the `Label` procedures.

The $\texttt{Scan}_i$ procedure returns a permutation π of the processes, such that the j'th location in the permutation is the rank of a timestamp reserved by process j among a set of timestamps, where each was reserved for the corresponding process at some time during the `Scan`. Again, the timestamps themselves are not directly observed by the calling procedure. (As before, the calls and returns to these procedures correspond in runs to local `Begin_Label`, `End_Label`, `Begin_Scan` and $\texttt{End\_Scan}(\pi)$ steps, where we add the permutation π returned by the `Scan` as a parameter to the `End_Scan` step.)

Figure 1 presents a natural, unbounded implementation of a concurrent timestamp system, *UCTSS*. In this implementation, the `Label` and `Scan` procedures access an array $Label[1,\ldots,n]$, of n integer variables, written by process i and read by all. Each process also maintains a local array $Temp_i[1,\ldots,n]$ of integer variables. The function $order(Temp_i)$ returns the permutation $(s_1,\ldots,s_n)$, where s_k is the lexicographic rank of the pair $(Temp_i[k],k)$ in the set $\{(Temp_i[1],1), \ldots, (Temp_i[n],n)\}$. (For example, $order((1001,1),(1500,2),(1500,3),(2564,4))$ is $(2,3,1,4)$.)

```
procedure $\texttt{Label}_i$;
  $\texttt{Begin\_Label}_i$
    forall $j \in \{1,\ldots,n\}$ do $Temp_i[j] := Label[j]$ od;
    if $\max_{j\in\{1..n\}}(Temp_i[j])) \neq Label[i]$ then $Label[i] := y$, where $y > x$;
  $\texttt{End\_Label}_i$;

procedure $\texttt{Scan}_i$;
  $\texttt{Begin\_Scan}_i$
    forall $j \in \{1,\ldots,n\}$ do $Temp_i[j] := Label[j]$ od;
  $\texttt{End\_Scan}_i(order(Temp_i))$;
```

Fig. 1. *UCTSS*: an unbounded concurrent timestamp system implementation.

The notation **forall** $j \in \{1, \ldots, n\}$ **do** S_j **od** means that the n distinct S_j statements may be executed in any order.

DEFINITION 3.1. *Any* concurrent timestamp system *is a collection of* `Label`$_i$ *and* `Scan`$_i$ *procedures and data structures satisfying the following two properties:*

Safety:. Each finite sequence of `Begin_Label`, `End_Label`, `Begin_Scan`, *and* `End_Scan` *steps that can occur when these procedures are called (provided only that the* `Label`$_i$ *and* `Scan`$_i$ *procedures are called sequentially by each process i) are also sequences that can occur when the procedures in Figure 1, are called in their place, regardless of the number of faults.*

Liveness:. In any run in which a process is not faulty and begins a `Label` *or* `Scan` *procedure, (takes a* `Begin_Label` *or* `Begin_Scan` *step), that procedure terminates (the process takes an* `End_Label` *or* `End_Scan` *step).*

These properties are the only properties of the `Scan` and `Label` procedures used in the proofs below.[2] Implementations of concurrent timestamp systems from bounded-size shared memory are presented in [?, ?, ?, ?, ?, ?]. In particular, Gawlick, Lynch and Shavit show specifically that the program sections and data structures of Figure 1 can be replaced with reads and writes of bounded-size shared memory, satisfying the safety and liveness properties above.[3]

Next, we state several straightforward properties of the `Label` and `Scan` procedures that will be used in the proofs below. In these proofs, as in the algorithm of the next section, we assume that no single process invokes more than one of these procedures concurrently.[4]

PROPOSITION 3.1. *The following are properties of any concurrent timestamp system, that hold in any run:*

(1) If j begins a `Label`$_j$ *procedure after i finishes a* `Label`$_i$*, then any* `Scan`*, by any process, performed entirely after both labeling procedures, and entirely before any subsequent labeling procedures by i, returns a permutation ordering i before j.*

(2) If a `Scan`$_i$ *procedure returns a permutation ordering i before j, then every* `Scan`$_i$ *procedure thereafter, until the start of the next* `Label`$_i$ *procedure (if any), returns a permutation ordering i before j.*

[2]The I/O automaton formalism is used in [?]. We have used a simpler, less general formalism to describe the specifics of ℓ-exclusion and our solution. Briefly, our model is easily cast in the I/O automaton formalism by mapping `Begin_section` and `End_section` to the input and output actions of the *CTSS* automaton of [?].

[3]Space- and time-efficient implementations of concurrent timestamp systems are a topic of active research. The Gawlick, Lynch and Shavit construction [?], using a snapshot primitive implementation from [?] as a subroutine, uses n registers of size $O(n^2)$, and each `Scan` or `Label` operation requires at most $O(n^2)$ primitive read and write operations. At the time of writing, the construction of a concurrent timestamp system from single-writer/multi-reader registers with the best time and space complexity is due to Dwork, Herlihy, Plotkin, and Waarts [?]. It uses n registers of size $O(n)$ and requires at most $O(n)$ reads and writes per `Scan` or `Label` operation.

[4]The properties in Proposition ?? may also be taken, together with the liveness property of concurrent timestamp systems, as an axiomatic specification of the `Label` and `Scan` procedures used in the ℓ-exclusion algorithm.

(3) *Suppose a* $\texttt{Label}_i$ *procedure* L_i *entirely precedes a* $\texttt{Label}_j$ *procedure, which in turn precedes a* $\texttt{Scan}_j$ *procedure* S_j*, where the* $\texttt{Scan}_j$ *returned a permutation ordering* j *before* k*. If* i *performs a new* $\texttt{Scan}_i$ *procedure that begins after* S_j *ends, without having executed a new* $\texttt{Label}_i$ *procedure since* L_i*, then the* $\texttt{Scan}_i$ *procedure returns a permutation ordering* i *before* k*.*

(4) *Suppose there is a set* C *of processes such that each* $i \in C$ *executes a* $\texttt{Label}_i$ *procedure,* L_i*, and (after some number of* $\texttt{Scan}_i$ *procedures) a* $\texttt{Scan}_i$ *procedure,* S_i*, but that there are no* $\texttt{Label}_i$ *procedures after* L_i*. (And the other processes are not constrained.) Let* $\texttt{Label}_j$ *be the* `Label` *operation which begins last, among the final* `Label`*s by processes in* C*, and let* $C' \subseteq C$ *contain those processes in* C *whose final* `Label` *operations end after the beginning of* $\texttt{Label}_j$*. (Hence* C' *contains at least process* j*.) Then there is a process* $i \in C'$ *such that the last* $\texttt{Scan}_i$ *procedure returns a permutation in which* i *is ordered after the other processes in* C*.*

(5) *Suppose some set* C *of processes are such that each* $i \in C$ *executes a* $\texttt{Label}_i$ *procedure,* L_i*, possibly followed by* $\texttt{Scan}_i$ *procedures, but with no later* $\texttt{Label}_i$ *procedures after* L_i*. (And the other processes are not constrained.) Let* L_j *be the* `Label` *procedure by a process in* C *that ends last. If each process* $i \in C$ *executes a* $\texttt{Scan}_i$ *procedure,* S_i*, that begins after* L_j *ends, then there is a process* $i \in C$ *such that* $\S_i$ *returns a permutation in which* i *is ordered before the other processes in* C*.*

The reader can verify that these are properties of runs of the *UCTSS* in Figure 1 (Proofs also appear in [?].) By the safety property, any finite sequence of `Begin_Label`, `End_Label`, `Begin_Scan` and `End_Scan` events of a concurrent timestamp system (providing each process runs operations sequentially) is also a sequence from a run of *UCTSS*. Hence, these properties hold for runs of arbitrary concurrent timestamp systems.

3.2 The ℓ-Exclusion Algorithm

The code of process i is described in Figure 2.

When entering the doorway, a process first raises a flag by setting x_i to true, then reserves a timestamp by executing the `Label` procedure, and finally records in S_i all the processes whose flags are set (i.e., that were not in the remainder) before i left the doorway. In the waiting room process i waits until the number of processes with an earlier timestamp, and which are neither in the remainder now and that were not in the remainder when i went through the doorway, is less than ℓ. Process i in the waiting room need not consider the timestamp of processes that were in the remainder when i left the doorway, since such processes might temporarily have an earlier timestamp, but by the time they next pass through the doorway, their timestamp will be later than that of i. After leaving the critical section, process i clears the flag x_i, signaling that it went back to the remainder. Finally, it performs an additional $\texttt{Label}_i$ procedure, so that when it resets x_i in its next pass through the doorway, the timestamp it holds will not be too early.

4. CORRECTNESS PROOF

LEMMA 4.1. *The algorithm in Figure 2, satisfies the* ℓ*-exclusion property.*

```
x_1, ..., x_n:                  shared variables
y_1, ..., y_n, S_i, L̄_i, Test_i:   local variables
repeat forever
0.   remainder_section_i

1.   x_i := true;                                        /* Begin_doorway_i */
2.   Label_i;
3.   S_i := ∅;
4.   forall j ∈ {1,...n} do
        if x_j = true then S_i := S_i ∪ {j} od;         /*End_doorway_i */

5.   repeat                                              /* Begin_waiting_room_i */
6.     forall j ∈ S_i do y_j := x_j od;
7.     L̄_i := Scan_i ;
8.     Test_i := {j ∈ S_i | y_j = true ∧ L̄_i[j] < L̄_i[i]} ;
9.   until (|Test_i| < l);                               /* End_waiting_room_i */

10.  critical_section_i

11.  x_i := false;                                       /* Begin_exit_i */
12.  Label_i;                                            /* End_exit_i */
end repeat
```

Fig. 2. A *First-In, First-Enabled* ℓ-exclusion algorithm: Code for process i.

Proof: By way of contradiction, assume that in a run ending in state s there is a set C of more than ℓ processes, each in the critical section. In this run, each process in C executed a `Label` and then at least one `Scan` procedure, and no later `Label` procedures. By part ?? of Proposition ??, there is a process $i \in C$ so that the last $\texttt{Scan}_i$ execution returned a permutation that ordered at least $|C|-1$ processes before i and no $\texttt{Label}_j$ for $j \in C$ comes entirely after the last $\texttt{Label}_i$. That is, the last $\texttt{Label}_j$ procedure (line 2) of each of the other processes j in C could not have started after i finished its $\texttt{Label}_i$ procedure in line 2. Hence, j finished its last execution of line 1 before i last started line 4, and when i executed line 4 it must have seen $x_j = true$. Finally, the value of x_j is *true* between j's leaving line 1 and j beginning line 11, after the critical section. Hence, every element of C other than i will be included in the set $Test_i$ computed by i in line 8, following its last $\texttt{Scan}_i$. But $|C|-1 \geq l$, contradicting the assumption that i entered the critical section. □

The next lemma argues that in consecutive executions of the **until** loop by process i, the size of $Test_i$ is monotonically non-increasing.

Lemma 4.2. *Let T^k and T^{k+1} be the values of $Test_i$ computed in two successive iterations of the waiting room loop by process i (without an intervening critical section or doorway). If $j \in T^{k+1}$, then $j \in T^k$.*

Proof: Assume that $j \in T^{k+1}$. Then $j \in S_i$ for the set S_i computed in the most recent preceding execution of the doorway, and in the $k+1$'st pass through the waiting room, the value of x_j is *true* and $\bar{L}_i[j] < \bar{L}_i[i]$. Suppose that $j \notin T^k$. Then in this k'th pass through the loop, i observed either x_j as *false* or $\bar{L}_i[i] < \bar{L}_i[j]$.

First consider the case that x_j was observed to be *false*. Then process i read x_j at least three times; once *true* (in the most recent preceding execution of line 4, since $j \in S_i$), then *false* and then (during the $k+1$'st pass) *true* again. In between these three observations of x_j, i did not perform any $\mathtt{Label}_i$ procedures. Then a write by j of $x_j := \mathit{false}$ must have occurred between the first two reads of x_j by i, and another write by j of $x_j := \mathit{true}$ must have occurred between the last two reads by i. But between these two writes by j a $\mathtt{Label}_j$ procedure by j occurred; that is, j must have finished a `Label` procedure before i read $x_j = \mathit{true}$ the final time, and this $\mathtt{Label}_j$ started after i read $x_j := \mathit{true}$, hence after i's last $\mathtt{Label}_i$ procedure. (See Figure 3.) By the first property in Proposition ??, $\bar{L}_i[i] < \bar{L}_i[j]$ in the subsequent $\mathtt{Scan}_i$'s, a contradiction.

		Process i		Process j
	2.	$\mathtt{Label}_i$		
	4.	$x_j = \mathit{true}$		
			11.	$x_j := \mathit{false}$
/* read for T^k */	6.	$x_j = \mathit{false}$	12.	$\mathtt{Label}_j$
			1.	$x_j := \mathit{true}$
/* read for T^{k+1} */	6.	$x_j = \mathit{true}$		
/* $\bar{L}_i[i] < \bar{L}_i[j]$ */	7.	$\mathtt{Scan}_i$		

Fig. 3. Sequence used in proof of Lemma 4.2.

It follows that in the k'th pass through the loop, i observed x_j as *true*. Since $j \notin T^k$, i must have observed $\bar{L}_i[i] < \bar{L}_i[j]$. But since no `Label` operations are performed in the waiting room loop, part ?? of Proposition ?? implies that i observes the same relation, $\bar{L}_i[i] < \bar{L}_i[j]$, in the $k+1$'st pass through the loop, a contradiction. □

LEMMA 4.3. *The algorithm in Figure 2 satisfies the ℓ-lockout avoidance property.*

Proof: Assume that in some infinite run there exists a set, *Stuck*, of $k \geq 1$ non-faulty processes that leave the remainder region and never re-enter it. Since these processes are non-faulty, they take an infinite number of steps outside the critical section. By the liveness property of concurrent timestamp systems, only a finite number of steps are ever taken inside any single `Scan` or `Label` procedure. Hence, the processes in *Stuck* fail the waiting room loop test infinitely many times. Then each process i in *Stuck* executed a final $\mathtt{Label}_i$ procedure before entering the loop, and then executed infinitely many `Scan` procedures, but no later `Label` procedures. (And the other processes are not constrained.) By part ?? of Proposition ??, there is a process $i \in \mathit{Stuck}$ such that a $\mathtt{Scan}_i$ procedure returns a permutation in which i is ordered before every other process j in *Stuck*. Moreover, part ?? of Proposition ?? implies every subsequent $\mathtt{Scan}_i$ procedure orders i before the other processes in *Stuck*. Hence, the processes in *Stuck* eventually do not contribute to the failure of i's loop test.

By part ?? of Proposition ??, if any process j runs a $\mathtt{Label}_j$ procedure that strictly follows i's last $\mathtt{Label}_i$ procedure, then every $\mathtt{Scan}_i$ that in turn follows this $\mathtt{Label}_j$ procedure will observe $\bar{L}_i[i] < \bar{L}_i[j]$ and hence thereafter j will not contribute to the failure of i's loop tests. Call the set of such j the *LateLabels*.

Finally, denote by *Remainder* the set of processes j that eventually enter the remainder section and never leave. Eventually, i will always observe $x_j = \mathit{false}$, and so the processes in *Remainder* will eventually stop contributing to the failure of i's tests.

Since i continues to fail the loop test, it follows that there are at least ℓ processes that are in neither *Stuck*, *LateLabels* nor *Remainder*. Then these processes are faulty–they either stop taking steps or take infinitely many steps in the critical section. □

LEMMA 4.4. *The algorithm in Figure 2 satisfies the first-in, first-enabled property.*

Proof: Let α be a finite system run ending in a state in which process i is in the waiting room and j is in the critical section, and suppose that i last left the doorway before j last entered it.

If i is not enabled to enter the critical section, then there exists a run extending α by an arbitrary number of steps of i (and perhaps of other processes), in which i does not enter the critical section. In particular, there is a run α' in which i executes an unsuccessful pass through the waiting room loop that takes place completely after j's entrance to the critical section. This requires at most n reads and a $\texttt{Scan}_i$ by i, to finish one unsuccessful pass through the loop that may be concurrent with j's entrance to the critical section, and then n reads and a $\texttt{Scan}_i$ by i to finish the second unsuccessful pass through the waiting room loop. (By the liveness property of concurrent timestamp systems, the $\texttt{Scan}_i$ procedures terminate.) The nonexistence of such a run would imply that i is enabled.

Choose such an extension α' of α that ends with the execution of line 8, $\mathit{Test}_i := \{j \in S_i \mid y_j = \mathit{true} \wedge \bar{L}_i[j] < \bar{L}_i[i]\}$, in this second unsuccessful pass through the waiting room loop. Let T_i be the value of Test_i in the final state of α', computed by i in this last pass through the loop, and let T_j be the value of Test_j in the final state of α. (Computed by j in its last pass, in α, through the waiting room loop before entering the critical section.) Since j passed the test and i did not, it follows that there is a process k in T_i that is not in T_j.

Following arguments similar to the proof of Lemma ??, we claim that if k is in Test_i, x_k must have the value *true* continuously between the last execution of line 4 by i (which read $x_k = \mathit{true}$) and the last read of x_k in line 6 by i (which also read $x_k = \mathit{true}$). Suppose this is not the case. Then as Figure 4 illustrates, a $\texttt{Label}_k$ occurs strictly after the last $\texttt{Label}_i$, and by part ?? of Proposition ??, $\bar{L}_i[i] < \bar{L}_i[k]$ after the subsequent $\texttt{Scan}_i$'s, contradicting the fact that k is in T_i.

Now we consider the relative order of events at Process i and j. (See Figure 5.) By assumption, i last left the doorway before j last entered it. Hence, the last execution of line 4 by i precedes the last execution of line 1 by j. Moreover, by the construction of α', j enters the critical section before the last read of x_k in line 6 by i. Therefore, the last read by j of x_k also takes place in the interval between the last execution of line 4 by i and the last read of x_k in line 6 by i. Hence, this read by j of x_k returns true, and k is in the set S_j in the final state of α, and in S_i in the final state of α'. Since k is not in T_j, it must be that $\bar{L}_j[j] < \bar{L}_j[k]$. Then by part ?? of Proposition ??, $\bar{L}_i[i] < \bar{L}_i[k]$, which would exclude k from T_i, a contradiction. □

(If the `Scan` and `Label` procedures are guaranteed to terminate in a bounded

		Process i	Process k
	2.	`Label`$_i$	
/* read for S_i */	4.	$x_k = true$	
			11. $x_k := false$
			12. `Label`$_k$
			1. $x_k := true$
/* read for T_i */	6.	$x_k = true$	
/* $\bar{L}_i[i] < \bar{L}_i[k]$ */	7.	`Scan`$_i$	

Fig. 4. Sequence relating processes i and k, used in proof of Lemma 4.4.

		Process i	Process j	
	1.	$x_i := true$		
	2.	`Label`$_i$		
/* $x_k = true$ between here... */	4.	$x_k = true$		
			1. $x_j := true$	
			2. `Label`$_j$	
			4. $x_k = true$	/* Read for S_j. */
			6. $x_k = true$	/* Read for T_j. */
			7. `Scan`$_j$	/* $\bar{L}_j[k] < \bar{L}_j[j]$ */
			10. `Begin_critical_section`$_j$	
/* ...and here. (Read for T_i.) */	6.	$x_k = true$		
/* $\bar{L}_i[k] < \bar{L}_i[i]$ */	7.	`Scan`$_i$		

Fig. 5. Sequence relating processes i and j, used in proof of Lemma 4.4.

number of steps, as in $UCTSS$ and the cited bounded implementations, the construction of α' in Lemma ?? requires extending α by only a bounded number of steps of i. Hence, the algorithm actually satisfies the stronger, bounded first-in, first-enabled condition defined in Section ??.)

5. OPEN PROBLEMS

Other generalizations of the mutual exclusion problem that allow several processes to execute separate critical sections concurrently are the Dining and Drinking Philosophers problems [?, ?, ?]. These problems involve multiple resources, each of which must be accessed in mutual exclusion. Thus, no two processes that share a common resource (represented by an edge in a process graph) enter the corresponding critical section at the same time. Each process must accumulate a sufficient set of resources in order to make progress. Given a primitive for accessing any single resource in mutual exclusion, the task is to coordinate processes so that the system does not deadlock, and each process eventually acquires all necessary resources. Since these problems are built on mutual exclusion primitives (as opposed to ℓ-exclusion), they typically cannot withstand even a single process fault. It would be interesting to explore a generalization which combines multiple resources each accessed in ℓ-exclusion.

Our formulation of the first-in, first-enabled condition may seem unsatisfactory, in that it conditions the early process's enabling on the later process's actual *entry* into the critical section, rather than its enabling. The following stronger definition might seem more natural:

DEFINITION 5.1. First-In, First-Enabled: *If i last left the doorway before j last entered it, i and j are in the waiting room and j is enabled, then i is enabled.*

Using the unbounded concurrent timestamp implementation *UCTSS*, our algorithm satisfies this property. However, we do not know of proof rules which allow this property to be demonstrated for arbitrary (specifically, bounded) implementations of the timestamp system. For example, a process i may become enabled passively when some other process j takes a step inside of the implementation of a $\texttt{Label}_j$ or $\texttt{Scan}_j$ procedure. In the corresponding run of *UCTSS*, i may not yet be enabled, or may already be enabled—the safety and liveness properties we assume do not relate the points at which processes are enabled.

Being enabled is a kind of branching property—whether it is true in a state depends on the possible futures of that state. Implementation relations based on language inclusion, such as we use to specify concurrent timestamp systems, do not preserve such properties. (The implementation may restrict nondeterminism, for example, so that processes are enabled in an implementation, but not in the corresponding state of the specification.) We need a specification theory that strengthens the implementation relation to include constraints on enabling, while maintaining an appropriate degree of abstraction (from particular implementations).

Self-stabilizing algorithms have the interesting property of converging to correct global states, no matter how they are initialized [?]. It would also be interesting to investigate self-stabilizing solutions to the ℓ-exclusion problem, and indeed, self-stabilizing variants of concurrent timestamp systems.

ACKNOWLEDGMENTS

The authors would like to thank the area editor and the referees, whose extensive, detailed comments and close reading of the paper led to many improvements.

REFERENCES

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In *Proc. 9th Annual ACM Symp. on Principles of Distributed Computing*, pages 1–13, August 1990. Also *Journal of the ACM*, to appear.
2. H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environment. In *Proc. of the 28th IEEE Annual Symp. on Foundations of Computer Science*, pages 337–346, October 1987.
3. K. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.
4. E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.
5. E. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.
6. E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
7. D. Dolev, E. Gafni, and N. Shavit. Towards a non-atomic era: ℓ-exclusion as a test case. In *Proc. 20th Annual ACM Symp. on the Theory of Computing*, pages 78–92, May 1988.
8. D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible! In *Proc. 21st Annual ACM Symp. on Theory of Computing*, pages 454–465, May 1989.
9. C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts. Time lapse snapshots. In *Proc. of the 1st Israel Symp. on Theory of Computing and Systems*, pages 276–290, May 1992.

10. C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible! In *Proc. 24th ACM Symp. on the Theory of Computing*, pages 655–666, May 1992.
11. M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Proc. 20th IEEE Annual Symp. on Foundations of Computer Science*, pages 234–254, October 1979.
12. M. Fischer, N. Lynch, J. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, January 1989.
13. R. Gawlick, N. Lynch, and N. Shavit. Concurrent time-stamping made simple. In *Proc. of the 1st Israel Symp. on the Theory of Computing and Systems*, pages 171–185. Springer-Verlag, May 1992.
14. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
15. A. Israeli and M. Pinhasov. A concurrent time-stamp scheme which is linear in time and space. In *Proc. of the 6th International Workshop on Distributed Algorithms*, pages 95–109, November 1992. Also, Technical Report, Technion, Haifa, Israel, March 1991.
16. H. Katseff. A new solution to the critical section problem. In *Proc. 10th Annual ACM Symp. on the Theory of Computing*, pages 86–88, May 1978.
17. L. Lamport. A new solution of *Dijkstra's* concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.
18. L. Lamport. The mutual exclusion problem. Part II: Statement and solutions. *Journal of the ACM*, 33(2):327–348, April 1986.
19. G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.
20. G. Peterson. Personal communication, 1988.
21. L. Rudolph. *Software Structures for Ultra-Parallel Computers*. PhD thesis, New York University, 1981.