Toward a Non-Atomic Era : ℓ-Exclusion as a Test Case

Danny Dolev* Eli Gafni† Nir Shavit‡

Abstract

Most of the research in concurrency control has been based on the existence of strong synchronization primitives such as test and set. Following Lamport, recent research promoting the use of weaker primitives, "safe" rather than "atomic," has resulted in construction of atomic registers from safe ones, in the belief that they would be useful tools for process synchronization. We argue that the properties provided by atomic operations may be too powerful, masking core difficulties of problems and leading to inefficiency. We therefore advocate a different approach, to skip the intermediate step of achieving atomicity, and solve problems directly from safe registers. Though it has been shown that "test and set" cannot be implemented from safe registers, we show how to achieve a fair solution to ℓ-exclusion, a classical concurrency control problem previously solved assuming a very powerful form of atomic "test and set". We do so using safe registers alone and without introducing atomicity. The solution is based on the construction of a simple novel non-atomic synchronization primitive.

1 Introduction

Understanding the fundamental complexities of synchronizing concurrent operations of processes that share resources has been a constant research topic in multi-process computing. Most of the past research was based on the assumption that even though processes access shared memory concurrently, they preform their operations atomically, or even worse, they have access to powerful constructs like atomic memory operations or "test-and-set." This assumption introduces the possibility of circularity in design

> – what value is there in breaking processes' collision using a primitive operation that itself requires breaking the same collision?

Powerful operations can mask the "core" difficulties in coordinating concurrent processes. Lamport has made an important step in avoiding the use of powerful operations by introducing a very weak shared memory communication mechanism, the Single-Writer-Single-Reader Safe Bit (subsequently referred to as a *Safe Bit*). Read

*IBM ARC and Computer Science Department, Hebrew University, Jerusalem.

†Computer Science Department, University of California, Los Angeles. Supported by NSF Presidential Young Investigator Award under grant DCR84-51396 and matching funds from IBM Faculty Development Award under grant D840622.

‡Computer Science Department, Hebrew University, Jerusalem. Supported by Graduate Student Award-Israeli Ministry of Communication, and by Leibnitz Fellowship-Hebrew University.

and Write operations on a Safe Bit are assumed to take up a non-zero interval of time whose actual length depends solely on the speed of the process performing the operation. A read interval that does not overlap a write interval, returns the last value that was written, otherwise it arbitrarily returns zero or one. The question we face is: what is the power of safe bits, and how many safe bits are used to solve a problem?

Much research has been directed towards constructing atomic registers from safe bits. That is, shared memory communication mechanisms in which, though performed concurrently, each operation can be considered to have been performed at an instance of time. These constructions [B87,BP87,N87,PB87,VA86,IL87] are costly (in the number of safe bits used), quite complex, and their correctness difficult to verify. Moreover, it has been shown by Herlihy [H87], that no "powerful" concurrency control element can be constructed using atomic registers, unless one process waits for another to complete its operation, (i.e. faster processes are forced to wait for a slower one to complete its operations). His conclusion is, that since the wait-freeness property of atomic registers is of no significance, one might as well employ a universal element that is "stronger" than test-and-set, even though it has the drawback of introducing waiting.

Our approach here, in contrast, is to tailor special waitfree data-structures to various classes of problems, in the belief that the special semantics and features of each class will allow one to manage with data-structures that are much less complex than atomic registers.

1.1 Our Results

The main result of the paper is the construction of a fair deterministic solution to the *ℓ-Exclusion problem* from safe bits directly, bypassing the construction of atomic registers. To this end, we introduce a *simple new data-type implemented from Safe Bits.* The cost of the solution (measured in number of safe bits used) is equivalent to the lowest cost [PB87, IL87] of constructing only a constant number of atomic registers.

The ℓ-exclusion problem, a classic example in concurrency control, was first introduced and solved by Fischer, Lynch, Burns, and Borodin in [FLBB79]. The problem arises when a group of processes are spontaneously invoked, possibly needing private access to one of ℓ identical resources. The ability of the solution to withstand the slow-down or even the crash of few processes ($\ell - 1$ of them), as well as the absence of collaboration of process not requesting a resource, are inherent to the problem. Previous solutions ([FLBB79,FLBB85]) to this problem, assumed existence of an atomic memory operation much more powerful than test-and-set, and were focused on achieving strong fairness properties. It was assumed that processes do not fail while performing this atomic operation.

A test-and-set operation is itself much more powerful than any operation that is implementable by safe bits. Its definition implies the mutual exclusion of the processes that concurrently access it. It can be used to serialize concurrent events by "time-stamping" them. One can easily reach a consensus among participating processes, in spite of a single crash, using a test-and-set. By a direct reduction to *Theorem 12* in [DDS87], one can prove that there is no implementation of test-and-set by atomic registers (by safe bits as well), even if only a single fault can occur. This result was previously proved in [LA87,CIL87] using a direct proof along the lines of [FLP85]. Thus, a solution to the ℓ-exclusion problem that does not employ "test-and-set", is of interest (let alone not using "atomic" registers, with their complex and costly implementation).

1.2 Properties of the Solution

In order to eliminate solutions that "hide" the waiting for a slow or faulty process to complete execution of concurrent operations, the failure model assumed is one in which a process may undetectably stop functioning while executing any operation in its protocol (this is the failure model of [FLBB79]). In fact, a process may fail while writing a single bit. Since processes can not distinguish between a failed process and a very slow

one, a process cannot "wait" on less than ℓ other processes.

Lamport [L86d] solves the mutual exclusion problem using safe registers, but in his solution a slow process can slow down (or block) every other process. He assumes a weak failure model in which a failed process eventually resumes its operations and ends up stopping gracefully. Thus, he avoids the problem that may be associated with a process "never" terminating its write operation.

The fairness we achieve in this paper is that any process that indicates its wish to utilize a resource, will eventually obtain it. Achieving fairness is the crux of the difficulty. Because there are ℓ resources, and because no process may wait for any other process since it might be faulty, resolving the contention among the processes in a fair manner becomes more complicated than in mutual exclusion, making a novel approach necessary for fair ℓ-exclusion.

1.3 A New Synchronization Data Structure

The basic entity required is a data-structure that implies some precedence relation between pairs of processes. Both processes have to be able to manipulate it, and all processes should be able to read it. The problem is, that if both processes may failstop while writing the data-structure, no single read outcome is ever possible without communication among all readers. What is crucial for the purpose of synchronization is that if only one process in a pair is faulty, the other process can still unambiguously manipulate the data structure to give precedence to the one that failed. This is accomplished with a data-structure that is implemented by a pair of symmetrical sub-structures, each consisting of three safe bits and manipulated by a single process. Our solution to the ℓ-exclusion problem utilizes three instances of the data structure per pair of processes, each in a different role. Employing the same data structure in different roles may indicate its usefulness in solving other aspects of synchronization problems, and may substitute atomicity in these applications.

The organization of our solution lends itself to a simple clear and modularly structured correctness proof. The problem is separated into its different elements. The interface between the elements is such that when one element is considered, the effect of the other elements can be abstracted via "black-boxes" whose interface is concise enough as not to increase the complexity of the solution of each element. All the separate solutions put together solve the original problem. A by-product of the ability of our solution to withstand $\ell - 1$ possible failures is that only the "slow-down" of ℓ processes or more can slow-down the progress of a process wanting a resource. Another implication is that the reading and writing of our data-structures, each in isolation, are waitfree and bounded, since no process needs to wait for any other.

In the following sections the ℓ-exclusion problem and its solution are presented. For clarity, some of the proofs are left to the appendix.

2 The Problem

A concurrent system is composed of n processes communicating via a shared memory consisting of of *safe registers* ([L86b]), the operations on which are reads and writes. Since *single-writer-multi-reader* (SWMR) boolean safe registers can easily be constructed from *single-reader-single-writer* boolean safe registers [L86b], it will be assumed that the shared memory consists of SWMR safe registers.

In the *ℓ-Exclusion Problem*, the program of every process consists of two distinguished sections: a *Remainder Section* and a *Critical Section*. Processes alternate between executing the remainder and the critical sections. A *failstopped* process may stop at any operation, or may never complete executing its current operation. The execution of any operation by a non-faulty process takes unbounded but finite time. It is assumed that failure is undetectable by other processes.

To solve the *ℓ-Exclusion Problem*, one is required to design *entry* and *exit* program sections to be performed before entering and after exiting the Critical Section, such that when added to the original program of every process, will assure that the following properties hold:

ℓ-Exclusion — no more than ℓ process are concurrently executing the critical section at any time.

ℓ-Deadlock Avoidance — if there always exists some non-faulty process outside the remainder, and less than ℓ processes failed outside the remainder, then, there always exists a non-faulty process that alternates between executing the Remainder and the Critical Sections infinitely often[1].

Lockout Freedom — if less than ℓ processes fail outside the Remainder, then any non-faulty process outside the Remainder will eventually execute the Critical Section.

In the paper definitions follow the basic system formalism of Lamport ([L86a]). A global time model of such a system is assumed. An abstract data type will be defined and proved to be implementable in the system. The protocols for solving the ℓ-exclution problem will be given in terms of the abstract data type.

3 The Solution

In [L86d], Lamport shows how fair solutions to mutual exclusion problems can be created, by superimposing a *fairness* construct on a completely unfair *deadlock-free mutual-exclusion* construct. Many such unfair *deadlock-free mutual-exclusion* constructs appear in the literature, where processes having greater ids or ones that are fast enough, may cause others to starve. To provide fairness, a two part *fairness* construct is added, one part of it to be performed before entering the unfair deadlock free exclusion construct, and the other after exiting the Critical Section. One is able to construct a solution based on such a superimposition, because the fairness construct can be allowed to prevent processes leaving the Remainder, from entering the unfair exclusion section, as long as some process is already in it (if there is no process already in it, all those entering have equal precedence, and favoring any of them will not impair the fairness). This possibility is unique to the *mutual* exclusion problem, since if some process is in the unfair exclusion construct when others enter the fairness construct, it has priority, and they can delay entering the Critical Section (via the unfair construct) without causing deadlock or violating fairness.

Unfortunately, the above type of modularity, is impossible for ℓ-exclusion, the reason being that there is more than one slot in the critical section. Having the fairness construct prevent all processes from entering the unfair exclusion construct because there is a process there, would cause ℓ-deadlock. Having it prevent only a necessary number would mean that it is not only a fairness construct but a solution to the problem. On the other, hand if it will allow processes to enter the unfair exclusion construct, then lockout may occur. A novel type of construction is therefore needed.

The solution presented in the sequel is of such a novel type, providing a different form of modularity than that described above. It decomposes (and is therefore presented) as follows: In *Subsection 3.2*, an ℓ-exclusion construct that is unfair and deadlock prone is presented. This construct is refined in *Subsection 3.3* to provide deedlock-free ℓ-exclusion. Though unfair, it will prevent the starvation of the non-faulty processes, outside the Remainder, that have the highest ids. In *Subsection 3.4*, a construct providing consistent *dynamic ids* is presented. This construct is embedded in the above constructions, so that any starved process will eventually obtain a dynamic id higher than any non-starved process, making itself eligible to pass through the unfair deadlock-free exclusion construct.

In a pictorial manner, the core of the above constructions may be viewed as a form of a "blackboard" in shared memory, where each process writes down its relations with others, for

[1] This definition is equivalent to the definition of *ℓ-Deadlock* in [FLBB79].

all to see. The board consists of a collection of abstract data-structures, each written by two processes, readable by all, and denoting a precedence relation between them. In the following section, a detailed definition of these abstract elements and their implementation is provided.

3.1 The Abstract Data-Types

In this section a novel abstract data type *fork* is defined and implemented, resembling the "fork" in Chandy and Misra's [CM84] solution to the Generalized Dining Philosophers problem. As in [CM84], it is used to establish precedence between the two processes that operate on it. The solution to the ℓ-Exclusion problem will employ three instances of the data-type per each pair of processes. In two of these instances, the full power of the fork will not be utilized, and a simpler data-type *arrow* is thus defined, by coalescing groups of states and groups of operations of the fork.

An instance of fork, $FORK_{ij}$, is associated with two processes i and j. Logically, it can be thought of as an actual "fork" that is shared by i and j. At all times the "fork" is "in the hand" of one of the processes. The fork cycles through four states in_use_i, $offered_i$, in_use_j, and $offered_j$, in that order. The transitions to in_use_i or $offered_i$ may happen as a result of the operations $take_i$ or $offer_i$ (respectively), executed by i, and likewise for j. An important characteristic of the fork is that it is "observable" by all processes (not only the two manipulating it)[2]. A $read_k$ operation may thus be performed by any process in the system, returning one of the four possible fork states (characterized by a collection of properties defined in the sequel). Formally:

Definition 1 *A fork $FORK_{ij}$ is a data structure that can be mutated by two processes, i and j (mutators), and read by all processes. Allowable operations on $FORK_{ij}$ are a $read_k(FORK_{ij})$*

[2] A major difference from the forks in [CM84], and the core difficulty in implementing it and proving its correctness.

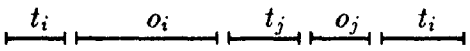

Figure 1: sequential fork manipulations

by an arbitrary k, and an $offer_k(FORK_{ij})$ or $take_k(FORK_{ij})$, by a mutator $k \in \{i,j\}$.

The first property of the fork is that each read returns just one of the four allowable states[3]

P1 "safeness": The value returned by a $read_k(FORK_{ij})$ operation is one of in_use_i, $offered_i$, in_use_j or $offered_j$.

The second property formalizes the independence between processes manipulating the fork, that is, that no operation requires the cooperation of other process.

P2 "wait-freeness": the read and mutation operations (by a process that does not fail before their completion) are completed within a finite time independent of the relative execution speeds of other processes.

From the above description, it might seem that operations on the fork are completely serial in time (as in *Figure 1*, where time runs from left to right, and the intervals represent the duration of *take* and *offer* operations). Following Lamport, one could provide serializability, i.e., the illusion that the operations are serial in time, by creating a fork that is atomic. It turns out though, that atomicity is not necessary. The reason for the simplicity of the fork's construction is that unlike atomicity, which is a claim about all points in time, the main claims made about the fork are restricted to specific points, or rather, to intervals of a limited type. The only claim that need be made about the fork at every point in time, is that it be safe! All other claims will be limited

[3] Note that this property in itself does not imply that the fork is "safe" [L86b], though this will follow when other properties are added.

to special types of intervals, such as those beginning at the end of the most recently started *take* or *offer* mutation, and ending before the start of the following mutation. As an example, given a situation as in *Figure 2*, claims about t_j (a $take_j$ operation) are limited to the interval c, bounded by the beginning of the next operation ($offer_j$).

Define $[R1..R2]$ to be the interval from the start of $R1$ to the end of $R2$, where $R2$ started after $R1$. Continuing in the description of the fork's properties, the following property states that the forks do not "change hands spontaneously." This is formalized by:

P3 "stability":

- **a.** Two *reads* $R1$ and $R2$ such that no mutation overlaps the interval $[R1..R2]$, return the same result.
- **b.** If a *read* $R1$ by mutator i returns either $offer_j$ or in_use_i, then any *read* $R2$ by mutator i that follows $R1$ such that the interval $[R1..R2]$ contains no mutation by i, will return the same result as $R1$.

A fourth property states that if some process saw a fork offered to j, j will also see this.

P4 "consistency": Let $R1$ be a *read* by a process k and $R2$ a *read* by mutator j that strictly follows $R1$. If $R1$ returned $offered_i$ and no mutation by j occur in the interval $[R1..R2]$, then $R2$ will return the same result.

Finally, the following fifth property enforces the power of the mutators to actually mutate the data-type.

P5 "mutability":

- **a.** A *read* $R2$ strictly following a sequence of a *read* $R1$ followed by $take_i$, where $R1$ returns $offered_j$ or in_use_i, will return in_use_i if no $offer_i$ started in the interval $[R1..R2]$.
- **b.** A *read* $R2$ strictly following a sequence of a *read* $R1$ followed by $offer_i$ (resp. $take_i$), where $R1$ returns in_use_i ($offered_j$), will not return in_use_i ($offered_j$) if one or more of $take_i$ ($offer_i$), $take_j$ and $offer_j$ does not occur in the interval $[R1..R2]$.
- **c.** A *read* $R2$ strictly following a sequence of a *read* $R1$ followed by $offer_i$, where $R1$ returns in_use_i, will return $offered_i$ if no mutation, other than $offered_i$, by either i or j occurs in the interval $[R1..R2]$.

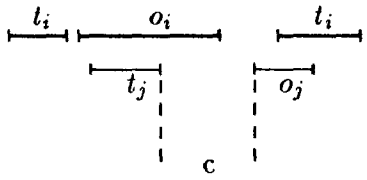

Figure 2: concurrent fork manipulations

Based on the above definitions of the fork data type, a possible approach one might take in implementing it would be as a mutual exclusion algorithm between i and j. The process gaining control of the critical section, would have the fork, releasing it only when it leaves the critical section. A problem though is that this solution is not *safe* (Where is the fork when both i and j are outside their critical sections?). One can overcome this problem easily if processes have access to some ordinary shared memory, that is, multi-reader multi-writer memory where a process can overwrite what others wrote (with the assumption of course, that it is safe, and that the user is responsible for providing mutually exclusive writing access to it). The improved implementation therefore uses a single safe bit F of this shared memory to represent the fork. For any two processes i and j, where without loss of generality $i > j$, $F = 1$ will mean that i has the fork, and $F = 0$ will mean that j has it. An additional SWMR safe bit w_i is used in every process i to provide mutual exclusion while processes access the shared memory bit F (since no such "ordinary" shared memory is available, F itself will later be constructed from safe bits). A

correctness proof of the construction appears in the Appendix.

Construction 1 *Let $FORK_{ij}$ be composed of a bit of shared memory, F, and two SWMR safe bits w_i, w_j, written by i and j respectively, where a $read_k(FORK_{ij})$ is performed as*[4]

if $F = 1$ **then**
  if w_i **then return** in_use_i
    else return $offered_i$ **fi**
else
  if w_j **then return** in_use_j
    else return $offered_j$ **fi**
fi;

and $take_i(FORK_{ij})$ and $offer_i(FORK_{ij})$ by a mutator i (those for mutator j are similar) are

$take_i$: **if** $read(FORK_{ij}) = offered_j$ **then**
  $w_i := true;$
  $F := 1;$
fi;
$offer_i$: **if** $read(FORK_{ij}) = in\_use_i$ **then**
  $w_i := false;$
fi;

then

Construction Lemma 1 *Construction 1 is a fork with properties* [P1] ... [P3].

To implement F by safe registers notice that since F is written only by the mutators i and j, it can be constructed from safe bits in a simple manner[5] (proof omitted).

Construction 2 *Let F be composed of two SWMR safe bits f_i and f_j, written by i and j respectively, where a read of F is*

if $f_i = f_j$ **then return** *1*
  else return *0* **fi;**

and a write of F (say of the value 1) is

if $f_i \neq f_j$ **then** $f_i :=$ **not** f_i
  else fi;

then

Construction Lemma 2 *Construction 1 with F as in construction 2 is a fork with properties* [P1] ... [P3].

How can one achieve "consistency" (property [P4] which is implied also by the stronger property [P5])? Given any variable v_i, and two disjoint subsets of processes $p, q \subset \{1 \ldots n\}$, consistency would mean that if processes in q see the new value written to v_i, processes in p reading v_i following the read of those in q, will also read the new value (unless a following write begins). A variable v_i *consistent with respect to* q is constructed in a simple manner (proof omitted):

Construction 3 *Let v_i be constructed of two variables vp_i and vq_i, where a write of v_i is performed as*

$vp_i := true;$
$vq_i := true;$

and a read of v_i is a read of vp_i for processes in p and of vq_i for processes in q, then[6]

Construction Lemma 3 *The variable v_i is consistent with respect to q.*

Based on the above, the following theorem can be proved (the proof appears in the Appendix).

Theorem 1 *Construction 1 with F as in Construction 2 and with w_i (and w_j) consistent with respect to $\{1 \ldots n\} - \{i\}$ ($\{1 \ldots n\} - \{j\}$) is a fork satisfying properties* [P1] ... [P5].

[4] The order of the reads of F and w_i, w_j is crucial!

[5] This use of "xor" bits appeared before in [P83],[L87d].

[6] Note that if one constructs a SWMR safe or regular register as in *construction 1* of [L86b], then consistency can be achieved using a single register by simply changing the ordering of the writes.

To simplify the presentation, the *fork* is further abstracted to create an *arrow* abstract data type. The arrow is actually a fork in which the states $offer_i$ and in_use_j are coalesced to form the state $i \rightarrow j$, and the same holds with the exchange of the roles of i and j. In a similar manner, the operations $redirect_i$ and $redirect_j$ are considered to be $take_i$ followed by $offer_i$, and $take_j$ followed by $offer_j$ respectively. Properties of the arrow will follow from those of the fork by way of its construction.

Construction 4 *The data structure $ARROW_{ij}$ is constructed from a $FORK_{ij}$ data structure mutable by i and j, where a $read_k(ARROW_{ij})$ operation for a process k is defined as*

case $read_k(FORK_{ij})$ **of**
  $offered_j$ **or** in_use_i: **return** $j \rightarrow i$;
  $offered_i$ **or** in_use_j: **return** $i \rightarrow j$;
end case;

and a $redirect_i(ARROW_{ij})$ as a sequence of a $take_i(FORK_{ij})$ followed by an $offer_i(FORK_{ij})$.

The following two claims about the abstract data types are made in order to simplify the proofs of the algorithms in following sections. The first claim characterizes a processes ability to manipulate the fork though the other process sharing it has failed. This includes for example the ability of a process to "pull an *offered* fork out of the other's hand".

Claim 1 *"Infinite mutation" – If a mutation by j lasts infinitely long, then*

- *if i performs $offer_i$ ($redirect_i$) infinitely often, then eventually all reads will never return in_use_i ($j \rightarrow i$);*
- *if i performs $take_i$ infinitely often and eventually does not perform any $offer_i$, then eventually all reads either always return in_use_i or never return in_use_i.*

Proof By [P2], j must have failed during its current infinitely long mutation, which is either $offer_j$ or $take_j$.

- Assume that i performs $offer_i$ infinitely often. Assume that following the failure of j there exists a time at which a processor performs a *read* $R1$ that returns in_use_i. Let $R2$ be any *read* strictly following an $offer_i$ that follows $R1$. By [P5b] and because j performs at most one of its operation during the interval $[R1..R2]$, $R2$ will not return in_use_i. (The claim for $redirect_i$ follows directly from the above).
- The proof follows directly by letting $R1$ of [P5a] be a *read* that returns in_use_i following the failure of j. ∎

The second claim deals with the conditions under which a a process that continuously tries to take the fork, will eventually obtain it.

Claim 2 *If all mutations apart from $offer_i$ are executed infinitely often while eventually no $offer_i$ is executed, then eventually all reads will return in_use_i.*

Proof Consider the following cases:

1. Eventually a process reads $offered_i$, then by [P5b], since no $offer_i$ will ever be performed, following the next $take_j$ no process will ever read $offered_i$ again.
2. Eventually a process reads in_use_j, then by [P5b], by exactly the same arguments no process will ever read in_use_j again.
3. Eventually a process reads $offered_j$ or in_use_i, then by [P5a], following the next $take_i$ all reads will always return in_use_i. ∎

3.2 ℓ-Exclusion

To obtain ℓ-exclusion, a *GRAPH* data structure is constructed. It consists of one instance of $ARROW_{ij}$ (denoted G_ARROW_{ij}) between every pair of processes $i, j \in \{1 \ldots n\}$. In addition to *GRAPH*, each process k maintains a SWMR safe bit x_k, which it sets to *true* upon leaving the remainder, and to *false* just before returning to it. The collection of all such x_k, $k \in \{1 \ldots n\}$ is denoted as X.

Every process wishing to execute the critical section, reads X and $GRAPH$. The order in which arrows in $GRAPH$ and x_k variables are read is unimportant[7]. By reading the $GRAPH$, a process obtains a tournament graph $G(i)$ on n nodes. Each edge (j,k) in the graph is directed in the direction read for G_ARROW_{jk}. It is important to note that $G(i)$ is a graph that *may never have existed*[8], since even the reading of a single edge (j,k) involves reading the six bits of a G_ARROW_{jk}, concurrently with possible redirection operations by i and j.

Let $R_i(G)$ denote the set of all nodes reachable via a directed path from a node i (including i itself) in a directed graph G. The result of the procedure $\Re(i, GRAPH, X)$ is defined as follows:

Definition 2 *$\Re(i, GRAPH, X)$: Read X and $GRAPH$. Let $G'(i)$ be the subgraph of $G(i)$ induced by all nodes k for which i read $x_k = true$. Return $R_i(G'(i))$.*

If the cardinality of the reachability set $\Re(i, GRAPH, X)$ returned to i is less than or equal to ℓ, node i may enter the critical section. The reason for choosing reachability is that a *transitive* precedence relation is needed. Taking "i is reachable from j" to mean "i is before j", if a process j is reachable from i, and a process k is reachable from j, then the transitivity of the reachability relation assures that both j and k will be before i. In general, transitivity assures that in any group of $\ell+1$ processes, some process will have all others before it[9]. Since $GRAPH$ is a dynamically changing structure, it remains to be shown that the reachability condition indeed suffices for ℓ-exclusion.

Observe that the reachability condition is to a large extent independent of the rules governing a process's mutation of arrows. In the construction below, this independence is abstracted by the procedure $oracle(GRAPH)$, whose arbitrary behavior will later be replaced by that of a deadlock prevention mechanism. Thus, $oracle(GRAPH)$ when called by i arbitrarily chooses some subset of arrows of which i is a mutator, and performs redirect on them. Let $redirect(i, j, GRAPH)$ be $redirect_i(G\_ARROW_{ij})$ and let every process $i \in \{ 1 \ldots n \}$ perform the algorithm that follows, then, even in face of the arbitrary behavior of $oracle(GRAPH)$, the following construction provides ℓ-exclusion:

Construction 5

do forever
  remainder
  $x_i := true$;
  for all j **in** $\{ 1 \ldots n \}$ **do**
    $redirect(i, j, GRAPH)$
  od;
L: oracle(GRAPH);
  if $|\Re(i, GRAPH, X)| > \ell$ **then goto** L **fi**;
  critical section
  $x_i := false$;
od;

then

Construction Lemma 5 *No more than ℓ processes will ever be in the critical section simultaneously.*

Proof Assume by a way of contradiction that a set C of more than ℓ processes is in the *critical section* between t^0 and t^1. Since no process in the critical section is in the middle of executing a mutation of $GRAPH$, then by [P1]-[P3], any $read_k(i,j)$ that started after t^0 and ended before t^1, for $i,j \in C$, will return a unique result for any k. Thus, the graph $G_t = (C, \{G\_Arrow(i,j) : i,j \in C\})$, as would have been defined by the above reads, is well defined. A contradiction will be obtained by showing that there exists a node $i \in G_t$ whose last execution of $\Re(i, GRAPH, X)$ before entering the Critical Section satisfies $|\Re(i, GRAPH, X)| \geq |R_i(G_t)| \geq \ell+1$.

[7] Although as mentioned before, the order of reads of single bits of each arrow is important.

[8] It is not even a snapshot [CL85], that is, one that could have existed.

[9] Unfortunately, the reachability relation is not a total order (which can be used to break deadlock), since it is not antisymmetric.

Observe that G_t is a tournament and consider the strongly-connected-component decomposition of G_t. There exists a "root" strongly connected component $R \subset G_t$, where for all $i \in R$ and $j \notin R$, $i \rightarrow j$. Let i be the last process in R to call $\Re(i, GRAPH, X)$ before entering the critical section, and let $t^* < t^0$ be the operation start time. From time t^* to t^1, no process in R executed any mutation, and therefore by [P3] the induced graphs of R in $G'(i)$ and G_t are isomorphic. Moreover, since any $read_i$ of an $G\_ARROW_{ij}, j \in C - R$ between t^0 and t^1 returns $i \rightarrow j$, and since i performed no mutation after t^*, it follows by [P3b] that the last read of $GRAPH$ by i returned $i \rightarrow j$.

Though this assures that i saw $R_i(G(i)) \geq \ell+1$, it still remains to be shown that $R_i(G'(i)) \geq \ell+1$. In its last read of $x_j, j \in C - R$, i must have read $x_j = true$, since otherwise j would have redirected its arrow to i following its last setting of x_j to true before entering the Critical Section. By [P5c], since i does not perform mutations after t^*, any read after t^0 would have returned $j \rightarrow i$. This would have contradicted the fact that the edge points from i to $j \in C - R$ in G_t. It thus follows that an edge $(i,j), j \in C - R$ in G_t implies an edge (i,j) in $G'(i)$. Thus, $R_i(G'(i)) \geq R_i(G_t) \geq \ell+1$. A contradiction. ∎

3.3 Deadlock Avoidance

In *construction 5*, deadlock may occur because many processes may repeatedly have reachability greater than ℓ, never entering the critical section. To overcome this problem, the arbitrary behavior of $oracle(GRAPH)$ is replaced by the rule that processes redirect arrows towards those with higher ids. If there were no faulty processes, it is easy to see that deadlock would be prevented, since in a group of blocked processes, the one with highest id would eventually have all arrows directed towards it, and therefore its reachability set would be of size less than ℓ.

Unfortunately this is not true if there is even a single faulty process. In the example of *Figure 3*,

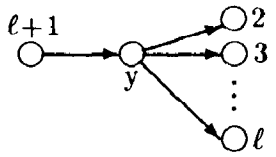

Figure 3: One Faulty Process

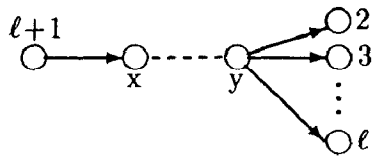

Figure 4: Several Faulty Processes

process y has an *id* less than all the other ℓ processes $2 \ldots \ell+1$ (say 1). If y fails with arrows directed as in the figure, even though the process $\ell+1$ with highest id will eventually have all arrows directed towards it by all non-faulty processes, its reachability will remain greater than ℓ. The reason for this is that processes $2 \ldots \ell$ will never redirect edges towards the smaller y, and y will never redirect the edge toward $\ell+1$ because it failed.

To overcome this problem, one can introduce the idea of redirecting arrows according to *induced ids*. The induced id of a process j as seen by i is the id of the process with highest id (excluding i) from which j is reachable in $G'(i)$. The problem occurring in the example of *Figure 3* is solved, since the induced id of y is $\ell+1$, and all processes will thus redirect their arrows towards it.

Yet, this modification does not suffice. In the example of *Figure 4*, the two processes x and y may have both failed in the middle of a *redirect* mutation. Thus, none of the properties assuring that all processes will read the same arrow state for G_ARROW_{xy} will ever hold. The largest live process with id $\ell+1$ may see the arrow pointed from x to y, and have reachability greater than ℓ, while all processes $2 \ldots \ell+1$ see the arrow directed from y to x, thus not seeing y as having an induced id of $\ell+1$.

The problem arises because the *induced id* "flows" from the inducing node through intermediate nodes that may be faulty. To overcome this problem, a second data structure *RANGE* is added, constructed from arrows in a manner equivalent to *GRAPH*, allowing the induced ids to "flow" directly. The induced id of j as seen by i is the id of the process with the highest id (excluding i) who points an arrow toward j in *RANGE*. Processes will indicate which processes are in their reachability sets by redirecting arrows in *RANGE* toward them. The following is a construction of a deadlock free ℓ-exclusion algorithm based on the above scheme.

Construction 6

```
do forever
   remainder
   x_i := true;
   for all j in { 1 ... n } do
            redirect(i, j, GRAPH)
   od;
L: update(GRAPH);
   update(RANGE);
   if |R(i, GRAPH, X)| > ℓ then goto L fi;
   critical-section
   x_i := false;
od ;
```

where the update procedures are

```
update(GRAPH):
   for all j do
      if not x_j then
        redirect(i, j, GRAPH) fi;
C:    if j > i and x_j then
        redirect(i, j, GRAPH);
        for all k do
            if j → k in RANGE then
                redirect(i, k, GRAPH)
            fi;
        od
      fi;
   od;

update(RANGE):
   for all j do
      if (j in R(i, GRAPH, X))
            or (not x_j) then
          redirect(i, j, RANGE);
      fi
   od ;
```

Construction Lemma 6 *The construction is free of ℓ-deadlock.*

In the next section the "static" ids used in line C of *construction 6* will be replaced by "dynamic" ones, therefore weaker requirements than the ones met by the static ids in *construction 6* are used in the proof below.

Proof Assume by a way of contradiction that the system is deadlocked. Let L be the set of live processes outside the remainder, and F be the set of faulty processes outside the remainder. There exists a time after which all live processes outside the remainder cease entering the Critical Section, and no new processes join L. Assume that eventually there exists a unique process with a maximal id *max* in L (for simplicity denote this process as *max*). A maximal id is such that all processes in L see themselves as having ids smaller than *max*, and *max* sees its id as larger than all other ids in L. (There always exists such a "static" id *max*). Since all processes $i \in L$ call $\Re(i, GRAPH, X)$ infinitely often, obtaining $|\Re(i, GRAPH, X)| \geq \ell+1$, and since by assumption $|F| \leq \ell - 1$, it must be the case that there exists a process $q \in L$ apart from *max* that appears in $\Re(max, GRAPH, X)$ of *max* infinitely often.

Without loss of generality, assume that there exists a fixed *path* of edges starting in *max* and leading to q, in which all the intermediate nodes (if they exist) belong to F, and that *path* appears infinitely often in $G'(max)$. Let $q_F \in F$ be the process that directly precedes q in *path*. Each time q_F appears in $\Re(max, GRAPH, X)$ when called by *max*, process *max* performs $redirect(max, q_F, RANGE)$. Since q_F is faulty and therefore does not start any new mutation, then by *Claim 1*, eventually the *RANGE* arrow between *max* and q_F is directed toward q_F. Also by *Claim 1*, every other live process reads it so.

Hence, eventually q will direct its *GRAPH* arrow toward q_F, and again because q_F is faulty, *Claim 1* implies the arrow will eventually stay that way in all reads. This contradicts the fact that *max* reads this arrow from q_F to q infinitely often. ∎

Corollary 2 Construction 6 *prevents lockout of the process with the highest id among the non-faulty processes outside the Remainder.*

Proof Notice that in the proof of *construction Claim 6*, one uses only the assumption that *max* does not enter the Critical Section. The proof proceeds verbatim even if the other non-faulty processes do enter the Critical Section infinitely often.

3.4 Avoiding Lockout

In this section a mechanism for creating dynamically increasing ids is presented. Using this mechanism, the ids of locked out processes can be made to increase, until they have an id higher than that of any process that is not locked out. By *corollary 2* , the dynamic *id* assignment grafted into the algorithm of the previous subsection will establish a lockout free ℓ-exclusion algorithm.

To create a *dynamic id mechanism*, an additional new data-structure *ID* is introduced, consisting of a collection of $FORK_{ij}$ data-types (denoted ID_FORK_{ij}), one for each pair processes $i,j \in \{1 \ldots n\}$, in a manner similar to that of *GRAPH*. Every process wishing to enter the Critical Section, will repeatedly attempt to collect all forks *offered* to it. The number of forks a process has *in_use* will constitute its dynamic id. The process will *offer* the forks it has *in_use* only after leaving the Critical Section. Thus, a process that is blocked and is repeatedly collecting forks, will have a monotonically increasing id.

To prove correctness of the mechanism, while abstracting the details of the previous constructions, define *oracle1* to be a procedure that arbitrarily generates a value of *loop* or *not-loop*, mimicking entrance to the Critical Section or failure to do so.

Construction 7 *Let ID be as defined below, and let every process $i \in \{1 \ldots n\}$ perform the following algorithm*

```
do forever
L: increment_your(ID);
   observe(ID);
   if oracle1 = loop then goto L fi ;
   initialize_your(ID);
   end;
od;
```

where increment_your(ID), observe(ID) and initialize_your(ID) are defined as

```
increment_your(ID):
  for all j in { 1 ... n } do
    take(i, j, ID) fi
  od;

initialize_your(ID):
  for all j in { 1 ... n } do
      offer(i, j, ID)
  od;

observe(ID):
  for all j in { 1 ... n } do
  count := 0
    for all k in { 1 ... n } do
      if read(j, k, ID) = in_use_j then
        count := count + 1
      fi
    od;
  id^j := (count, j);
  od;
```

Construction Lemma 7 *If there exists a non-faulty process i, that in an infinite run has ceased performing initialize_your operations, then*

1. *All live processes will eventually have id^i greater than all id^j for processes that initialize_your infinitely often.*
2. *All live processes will eventually have the same value for id^i.*

Proof Let B be the set of "blocked" live processes which from some time on do not perform *initialize_your*, U the set of "unblocked" live processes which perform it infinitely often, and F the set of faulty processes. Eventually every process in $j \in U$ performs *take* followed by *offer* infinitely often, and every process in $i \in B$ performs *take* infinitely often and never performs *offer*. Thus, by Claim 2, eventually all ID_FORKs between processes in B and processes in U must be *read* as *in_use* at B. In addition, since every process in U performs *offer* infinitely often and every process in F is either forever in the midst of the same mutation or not mutating forever, then again by *Claim 1*, eventually no *read* of an ID_FORK between a process in U and a process in F returns $in\_use_k, k \in U$. It follows that eventually each id in B will be *read* as being greater or equal to $|U|$, and each id in U will be less than $|U|$, which establishes the first part of the lemma.

The second part of the lemma follows directly from the first part and *Claim 1.* ∎

Combining the above constructions, the following is a solution to the ℓ-Exclusion Problem.

Construction 8

```
do forever
   remainder
   x_i := true;
   for all j in { 1 ... n } do
            redirect(i, j, GRAPH)
   od;
L: increment_your(ID);
   observe(ID);
   update(GRAPH);
   update(RANGE);
   if |ℜ(i, GRAPH, X)| > ℓ then goto L fi;
   critical-section
   initialize_your(ID);
   x_i := false;
od ;
```

Construction Lemma 8 *The construction provides lockout free ℓ-exclusion.*

Proof Follows from Construction Lemmas 6 and 7 and corollary 2. ∎

4 Acknowledgements

We wish to thank Mike Merrit and Larry Rudolph for important conversations during the course of our work.

5 References

[B87] B. Bloom, "Constructing two-writer atomic registers," *Proc. 6th ACM Symp. on Principles of Distributed Computation*, 1987, pp. 249-259.

[BP87] J. E. Burns, and G. L. Peterson, "Constructing two-writer atomic registers," *Proc. 6th ACM Symp. on Principles of Distributed Computation*, 1987, pp. 222-231.

[CIL87] B. Chor, A. Israeli, and M. Li, "On Processor Coordination Using Asynchronous Hardware", *Proc. 6th ACM Symp. on Principles of Distributed Computation*, 1987, pp. 86-97.

[CL85] K. M. Chandy, and L. Lamport, "Distributed Snapshot: Determining Global States of Distributed Systems," *ACM Trans. on Prog. Lang. and Sys. 1, 6* 1985, pp. 63-75.

[CM84] K. M. Chandy, and J. Misra, "The Drinking Philosophers Problem," *ACM Trans. on Prog. Lang. and Sys. 6, 4* 1984, pp. 632-646.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM 34*, 1987, pp. 77-97.

[FLBB79] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, "Resource Allocation with Immunity to Limited Process Failure," *Proc. 20th IEEE Symp. on Foundations of Computer Science*, 1979, pp. 234-254.

[FLBB85] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, "Distributed Fifo Allocation of Identical Resources Using Small Shared Space," *MIT/LCS/TM-290,* 1985.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Processor," *J. ACM 32*, 1985, pp. 374-382.

[H87] M. P. Herlihy, "WaitFree Implementations of Concurrent Objects," Technical Report, Dept. of CS, CMU, 1987.

[IL87] A. Israeli, and M. Li, "Bounded Time-Stamps," *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, 1987, pp. 371-382.

[L86a] L. Lamport, "On Interprocess Communication. Part I: Basic Formalism," *Distributed Computing 1, 2* 1986, 77-85.

[L86b] L. Lamport, "On Interprocess Communication. Part II: Algorithms," *Distributed Computing 1, 2* 1986, pp. 86-101.

[L86c] L. Lamport, "The Mutual Exclusion Problem. Part I: A Theory of Interprocess Communication," *J. ACM 33, 2* 1986, pp. 313-326.

[L86d] L. Lamport, "The Mutual Exclusion Problem. Part II: Statement and Solutions," *J. ACM 33, 2* 1986, pp. 327-348.

[LA87] M. G. Loui, and H. Abu-Amara, "Memory Requirements for Agreement Among Unreliable Asynchronous Processes", *Advances in Computing Research*, vol. 4, 1987, pp. 163-183.

[N87] R. Newman-Wolfe, "A Protocol for Wait-free Atomic, Multi Reader Shared Variables," *Proc. 6th ACM Symp. on Principles of Distributed Computation*, 1987, pp. 232-248.

[P83] G. L. Peterson, "Concurrent Reading While Writing," *ACM TOPLAS 5, 1* 1983, pp. 46-55.

[PB87] G. L. Peterson, and J. E. Burns, "Concurrent Reading While Writing," *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, 1987, pp. 383-392.

[VA86] P. Vitanyi, and B. Awerbuch, Atomic shared register access by asynchronous hardware, *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986, pp. 233-243.

6 Appendix

In this appendix, an informal proof of the validity of constructions 1-3 is presented. A formal proof based on Lamport formalism ([L86a]) will be given in a later version of the paper.

For the proof consider a given $FORK_{ij}$. It will be argued that the set of lower level system executions defined by constructions 1-3 are implementations of a set of higher level system executions defined by the abstract data type. It is assumed that no system execution begins in the middle of a mutation. A global time model of system executions is assumed.

Assume that initially any read returns $F = 0$ and $w_i = w_j = false$.[10]

Consider the following *pre-conditions* and *post-conditions* for mutation operations. A condition such as $\{ F = 0 \wedge w_j = true \}$ is interpreted to mean that a read by any process would return $F = 0$ and $w_j = true$. These conditions will be required to hold only for reads performed in intervals of the designated type described in section 3.1. They constrain possible alternative executions of the prefixes of sequences of the lower

[10]This assumption is not necessary but simplifies the proofs.

level operations as implied by the implementation. Thus, if a read is performed in an interval as designated, the appropriate value will be returned. The conditions are written in short form, where for a program statement S and some predicate P, $\{condition\}S\{post\_condition\}$ means that $\{condition \wedge P\}S\{post\_condition \wedge P\}$ and $\{\neg condition \wedge P\}S\{\neg condition \wedge P\}$ (where *condition* and *post_condition* are different).

$take_i$:
$\{F = 0 \wedge w_j = false\}$
if $read(FORK_{ij}) = offered_j$ **then**
  w_i := true;
$\{F = 0 \wedge w_i = true\}$
  F := 1;
$\{F = 1 \wedge w_i = true\}$
fi;

$offer_i$:
$\{F = 1 \wedge w_i = true\}$
if $read(FORK_{ij}) = in\_use_i$ **then**
  w_i := false;
$\{F = 1 \wedge w_i = false\}$
fi;

$take_j$:
$\{F = 1 \wedge w_i = false\}$
if $read(FORK_{ij}) = offered_i$ **then**
  w_j := true;
$\{F = 1 \wedge w_j = true\}$
  F := 0;
$\{F = 0 \wedge w_j = true\}$
fi;

$offer_j$:
$\{F = 0 \wedge w_j = true\}$
if $read(FORK_{ij}) = in\_use_j$ **then**
  w_j := false;
$\{F = 0 \wedge w_j = false\}$
fi;

Given the initial conditions, the only mutation whose pre-condition is met is $take_i$. Since the pre-condition of any mutation that could be concurrent with $take_i$ will not hold until the completion of $take_i$, the post conditions of $take_i$ will hold upon its completion.

After the completion of the $take_i$ mutation, the pre-condition for $offer_i$, and only for it, holds. Until the execution of the assignment to w_i in $offer_i$, the pre-conditions of none of the other mutations hold. Only once this assignment operation is started, the pre-condition of $take_j$, and only $take_j$, may hold. Thus i might be writing w_i while j performs $take_j$, yet, this does not impair the correctness of the post-condition of $take_j$, given that its pre-condition was *true*. Though the post condition of i might not hold following the a $take_j$ (and only it), it will not matter since anyhow it is a pre-condition only for $take_j$. The pre-condition of any mutation by i will not hold prior to the assignment of w_j in $offer_j$. This will only happen after the completion of the current $take_j$, after which only the pre-condition of an $offer_j$ can hold, and so on.

The above arguments informally imply that once a pre-condition of a mutation holds, it will continue to hold until the mutation takes place, and as long as it doesn't take place the pre-conditions for all other mutations do not hold.

Proof of Construction Lemma 1 Properties P1, P2, and P3a clearly hold. The above arguments imply that when the pre-condition for a mutation holds, it will hold until the process performs the mutation, therefore Property P3b holds. ∎

Proof of Theorem 1 Construction Lemma 3 implies that if any process reads $offered_i$, then j also will also read it. Thus, the pre-condition to $take_j$ holds and by the above arguments will still hold as long as j will not perform $take_j$. Therefore, P4 holds.

Property P4 further implies that once a process has read $offered_j$ following which a $take_i$ was performed, every process will read in_use_i as long as an $offer_i$ will not start.

Proofs of all other properties follow by similar arguments. ∎