Computing with Infinitely Many Processes[*]

Michael Merritt[†] Gadi Taubenfeld[‡]

December 5, 2013

Abstract

We explore four classic problems in concurrent computing (election, mutual exclusion, consensus, and naming) when the number of processes which may participate is unbounded. Partial information about the number of processes actually participating and the concurrency level is shown to affect the computability and complexity of solving these problems when using only atomic registers. We survey and generalize work carried out in models with known bounds on the number of processes, and prove several new results. These include improved bounds for election when participation is required and a new adaptive starvation-free mutual exclusion algorithm for unbounded concurrency. We also survey results in models with shared objects stronger than atomic registers, such as test&set bits, semaphores or read-modify-write registers, and update them for the unbounded case.

[*]A preliminary version of the results presented in this paper, appeared in *Proceedings of the 14th International Symposium on Distributed Computing*, LNCS 1914:164-178, October 2000.

[†]AT&T Labs, 180 Park Ave., Florham Park, NJ 07932-0971. `mischu@research.att.com`.

[‡]The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel. `tgadi@idc.ac.il`.

1 Introduction

In most work on the design of shared memory algorithms, it is assumed that the number of processes is bounded and *a priori* known. Here we investigate the design of algorithms assuming no *a priori* known bound on the number of processes. In particular, we assume that, in an infinite run, the number of active processes may be unbounded. A process is active in a given run if it has taken at least one step in that run. The primary motivation for such an investigation is to understand the limits of distributed computation–in particular, whether bounding the number of active processes is necessary in order to solve specific distributed problems. While, in practice, the number of processes will always be bounded, algorithms designed for an unbounded number of processes may scale well: their complexity may depend on the actual contention and not on the total number of processes.

An important factor in designing algorithms where the number of processes is unknown, is the *concurrency level,* the maximum number of processes that may be active simultaneously, participating in the algorithm at the same instant of time. The actual concurrency in a given run is often called *point contention.* We consider computability when the concurrency level imposes an *a priori* bound on the point contention. (A weaker notion of concurrency, *interval contention*, is not considered here.) We distinguish between the following concurrency levels:

- *known bounded:* There is a known bound (denoted by c) on the maximum number of processes that are simultaneously active, over all runs.
- *unknown bounded:* In each run, there is a bound on the maximum number of processes that are simultaneously active.
- *unbounded:* In each run, the number of processes that are simultaneously active in any state is bounded but can grow without bound.

A *fault-free* model refers to a model where processes never fail. We will design algorithms both for the fault-free model, and for models where processes may fail by crashing. When assuming a fault-free model with *required participation* many problems are solvable using only constant space. Required participation means that every process must eventually execute its code. Required participation had been considered, for example, for the symmetric model with a known bound on the number of processes, by Styer and Peterson [SP89]. However, a more interesting and practical situation is one in which participation is not required, as is more usually assumed when solving resource allocation problems. For example, in the mutual exclusion problem, a process can stay in the reminder region forever and is not required to try to enter its critical section. Next, we consider at the case, where *some* of the processes, but not necessary all of them, must participate.

We use the notation $[\ell, u]$*-participation* (where u might be ∞) to mean that at least ℓ and at most u processes participate. Thus, for a total of n processes, $[1, n]$-participation is the same as saying that participation is not required, while $[n, n]$-participation is the same as saying that participation is required. Requiring that all processes must participate does not mean that there must be a point at which they all participate at the same time. That is, the concurrency level might be smaller that the upper bound on the number participating processes. Notice also that if an algorithm is correct assuming $[\ell, u]$-participation, then it is also correct assuming $[\ell', u']$-participation, where $\ell \leq \ell' \leq u' \leq u$. When n is ∞, $[\infty, \infty]$-participation means that an infinite number of processes must participate.

We consider the following problems:

Mutual exclusion: The mutual exclusion problem is to design a protocol that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65]. Each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections of code: the *remainder, entry, critical section* and *exit.* A process starts by executing the remainder code. At some point the process might need to execute some code in its critical section. In order to access its critical section, a process has to execute the entry code, which guarantees that while it is executing its critical section, no other process is allowed to execute its critical section. In addition, once a process finishes its critical section, the process executes its exit code, in which it notifies other processes that it is no longer in its critical section. After executing the exit code, the process returns to the remainder.

The mutual exclusion problem is to write the *entry code* and the *exit code* in such a way that the following two basic requirements are satisfied.

Mutual exclusion: No two processes are in their critical sections at the same time.

Deadlock-freedom: If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.

The deadlock-freedom property guarantees that the system as a whole can always continue to make progress. However deadlock-freedom may still allow "starvation" of individual processes. That is, a process that is trying to enter its critical section, may never get to enter its critical section, and wait forever in its entry code. A stronger requirement, which does not allow starvation, is defined as follows.

Starvation-freedom: If a process is trying to enter its critical section, then this process must eventually enter its critical section.

In solving the problem, it is assumed that once a process starts executing its critical section, the process always finishes it regardless of the activity of the other processes. Of all the problems in interprocess synchronization, the mutual exclusion problem is the one studied most extensively. This is a deceptive problem, and at first glance it seems very simple to solve.

Consensus: The consensus problem is to design a protocol in which all correct processes reach a common decision based on their initial opinions [FLP85]. More formally, the problem is defined as follows. There are n processes $p_1, p_2, \ldots, p_n$. Each process p_i has an input value $x_i \in \{0, 1\}$. The requirements of the consensus problem are that there exists a *decision value* v such that: (1) each non-faulty process eventually decides on v, and (2) $v \in \{x_1, x_2, \ldots, x_n\}$. In particular, if all input values are the same, then that value must be the decision value.

Election: The (leader) election problem is to design a protocol where every correct process eventually decides on a value in $\{0, 1\}$, and exactly one process (the leader) decides on the value 1 [FL87]. It is not required that the processes know the identity of the elected leader. (In a model with an unbounded number of processes, identifying the leader obviously requires infinite space–the weaker $\{0, 1\}$ formulation makes the lower bounds more complex.) To prevent trivial solutions, we assume that a process does not initially know the identifiers of the other potential participants. This assumption prevents us from having a solution for the election problem where all processes choose process 1 as a leader without even needing to communicate (because, it may be the case that there is no participating process with identifier 1).

Naming: The naming problem is to assign unique names to initially identical processes [ABNDPR90]. A solution to the problem is required to be *wait-free*, that is, it should guarantee that every participating process will always be able to get a unique name and terminate in a finite number of steps regardless of the behavior of other processes. After acquiring a name a process may later release it. A process terminates only after releasing the acquired name. This problem is different from the renaming problem, which allows processes, with distinct initial names from a large name space, to get distinct new names from a small name space.

Below we give a short summary of our results. The notions of adaptive algorithms, symmetric algorithms and wait-free algorithms are formally defined in Section 2.2.

We show that even with a known bound on concurrency and required participation, election (using registers) requires infinite shared space. Among the novel algorithms presented are two demonstrating that either a single shared register of infinite size, or infinitely many shared bits, suffice for both election and consensus. (In addition, the first algorithm is adaptive.) If, in addition, test&set bits are used, then solving the above problem requires only finite space.

When the concurrency level is unknown bounded, we show that using an infinite number of registers is a necessary and sufficient condition for solving the mutual exclusion problem. (The algorithm presented is adaptive, symmetric, and satisfies starvation-freedom.) We then show that even infinitely many test&set bits do not suffice to solve naming assuming unbounded concurrency. However, naming can be solved assuming unknown bounded concurrency using test&set bits only, hence it separates unknown bounded from unbounded concurrency.

The tables below summarize the results discussed in this paper. As indicated, many are derived from existing results, generally proven for models where the number of processes is bounded and a priori known. We use the following abbreviations: DF for deadlock-free; SF for starvation-free; mutex for mutual exclusion; RW for atomic read/write registers; T&S for test&set bits; wPV for weak semaphores; RMW for read-modify-write registers; U for Upper bound; L for Lower bound. For the case of the election problem using atomic registers, we have two different lower bounds, denoted L_1 and L_2, in the table, the first concerns the number of registers, and the other concerns their size. Similarly, for SF mutex, we have two different upper bounds, denoted U_1 and U_2, in the table. (The default is "No" for the adaptive and symmetric columns. All lower bounds hold for the most general, non-adaptive and asymmetric case.) An earlier version of this paper appeared in [MT2000].

Problem			Model		Result					
	Name	Bo-und	Concur-rency $c > 1$	Partici-pation	Properties		Space		Thm #	Using results from
					adap-tive?	sym-metric?	#	size		
1	Election	L_1	c	$[c, \infty]$			∞ state space		3.1	
		U	c	$[c, \infty]$			∞	2	3.2	
	(Upper	U	2	$[2, \infty]$	Y	Y	1	∞	3.3	
	bounds	U	c	$[c, \infty]$	Y	Y	2	∞	3.4	
	also hold	U	c	$[c, \infty]$	Y		1	∞	3.5	
	for	U	c	$[1, \infty]$		Y	$\log c + 1$	∞	4.1	[SP89]
	consensus)	L_2	c	$[1, \infty]$			$\log c + 1$	2	4.1	[SP89]
2	DF mutex	U	c	$[1, \infty]$		Y	c	∞	4.1	[SP89]
		L	c	$[1, \infty]$			c	2	4.1	[BL93]
3	Election, SF mutex	L	unknown bounded	$[1, \infty]$			∞	2	4.1	$1L_2$,2L
		U_1	unbounded	$[1, \infty]$	Y	Y	∞	∞	4.2	
		U_2	unbounded	$[1, \infty]$			∞	2	4.7	
4	Implement Test & Set	L	unknown bounded	$[1, \infty]$			∞	2	5.1	3L,5U
		U	unbounded	$[1, \infty]$	Y	Y	∞	∞	5.1	$3U_1$
		U	unbounded	$[1, \infty]$			∞	2	5.1	$3U_2$

Results using atomic registers in a fault-free model

Problem			Model				Result					
	Name	Bo-und	Concur-rency $c > 1$	Partici-pation	Fault tole-rance	Shared objects	Properties		Space		Thm #	Using results from
							adap-tive?	sym-metric?	#	size		
5	Election, DF mutex	L	c	$[\infty, \infty]$	0	T&S			1	2		trivial
		U	unbounded	$[1, \infty]$	∞	T&S	Y	Y	1	2		trivial
6	SF mutex	L	unknown bounded	$[1, \infty]$	0	RW,T&S			∞	2	5.2	[Pet94]
		U	unbounded	$[1, \infty]$	0	RW			∞	2	4.7	$3U_2$
7	Naming	U	unknown bounded	$[1, \infty]$	∞	T&S	Y	Y	∞	2	5.3	[AT96] [MA95]
		L	unknown bounded	$[1, \infty]$	∞	T&S	Y	Y	∞	2	5.3	
		L	unbounded	$[1, \infty]$	∞	T&S			no alg		5.3	
8	Consen-sus	L	2	$[\infty, \infty]$	0	RMW			2	2		trivial
		U	unbounded	$[1, \infty]$	∞	RMW	Y	Y	1	3		trivial
9	SF mutex	L	unknown bounded	$[1, \infty]$	0	RMW			1	∞	6.1	[F+89]
		U	unbounded	$[1, \infty]$	0	RMW	Y	Y	1	∞	6.1	[B+82]
		U	unbounded	$[1, \infty]$	0	RW			∞	2	4.7	$3U_2$
10	SF mutex	U	unbounded	$[1, \infty]$	0	RW,wPV	Y	Y	2,2	2	6.2	[FP87]

Results using stronger shared objects

The remainder of the paper is organized as follows: In Section 2, we present a formal model of concurrent systems, and define various properties and complexity measures that we shall use. In Section 3, we demonstrate that requiring a minimum number of processes to participate is a powerful enabling assumption. We begin by showing that any solution to election, for unbounded number of processes, requires infinite shared space. Then, we show that election and consensus can be solved either by using an infinite number of binary registers, or a single register of infinite size. In Section 4, we present an adaptive starvation-free mutual exclusion algorithm for unbounded concurrency using atomic registers. In section 5, we solve the starvation-free mutual exclusion problem and the naming problem, using atomic registers and test&set bits. In section 6, we consider the space complexly of solving the mutual exclusion problem, using RMW bits and semaphores. Related work is discussed in Section 7. We conclude with Section 8.

2 Underlying system model and basic definitions

2.1 Asynchronous systems

Our model of computation consists of an asynchronous collection of n processes that communicate via shared objects. A process executes a sequence of steps as defined by its algorithm. A process executes correctly its algorithm until it (possibly) crashes. After it has crashed a process executes no more steps. Given a run, a process that crashes is said to be *faulty* in that run, otherwise it is *correct.* In an asynchronous system there is no way to distinguish between a faulty process and a process that is very slow. A process is *active* (or participate) in a given run if it has taken at least one step in that run.

An *event* corresponds to an atomic step performed by a process. For example, the events which correspond to accessing registers are classified into two types: read events which may not change the state of the register, and write events which update the state of a register but does not return a value. We use the notation e_p to denote an instance of an arbitrary event at a process p.

A *run* is a pair $(f,\ R)$ where f is a function which assigns initial states (values) to the objects and R is a finite or infinite sequence of events. When R is finite, we say that the run is finite. An implementation of an object from a set of other objects, consists of a non-empty set C of runs, a (possibly infinite) set N of processes, and a set of shared objects O. For any event e_p at a process p in any run in C, the object accessed in e_p must be in O. Let $x = (f,\ R)$ and $x' = (f',\ R')$ be runs. Run x' is a *prefix* of x (and x is an *extension* of x'), denoted $x' \leq x$, if R' is a prefix of R and $f = f'$. When $x' \leq x$, $(x - x')$ denotes the suffix of R obtained by removing R' from R. Let $R; T$ be the sequence obtained by concatenating the finite sequence R and the sequence T. Then $x; T$ is an abbreviation for $(f,\ R; T)$.

The current *value* (state) of an object (or of an algorithm) at the end of a finite run is determined by the values that were written into the memory of that object, and its initial state (determined by f). Process p is *enabled* at run x if there exists an event e_p such that $x; e_p$ is a run. For simplicity, we write xp to denote either $x; e_p$ when p is enabled in x, or x when p is not enabled in x. Register r is a *local* register of p if only p can access r.

For any sequence R, let R_p be the subsequence of R containing all events in R which involve p. Runs $(f,\ R)$ and $(f',\ R')$ are *indistinguishable* for a set of processes P, if and only if for all $p \in P$, $R_p = R'_p$ and $f(r) = f'(r)$ for every local register r of p. It is assumed that the processes are deterministic, that is, if $x; e_p$ and $x; e'_p$ are runs then $e_p = e'_p$.

The runs of an asynchronous algorithm must satisfy the following properties:

1. If a *write* event which involves p is enabled at run x, then the same event is enabled at any finite run that is indistinguishable to p from x.
2. If a *read* event which involves p is enabled at run x, then some read event of the same object is enabled at any finite run that is indistinguishable to p from x. That is, if p is “ready to read” some value from an object then an event on some other process cannot prevent p from reading from this object although p may read a different value.

3. It is possible to read only the last value that was written into an object or its initial value if the object has not been written yet. A read event has no effect on the state.

It follows from the above properties that if an event e_p is enabled at a run x, then e_p is enabled at any run y that is (1) indistinguishable to p from x, and (2) the state of the object accessed during e_p is the same in x and y.

In the sequel, we use a function, called $Enum()$, from the positive integers into process id's. $Enum(x)$ returns the x-th process id in an infinite enumeration. We require that the function $Enum()$ satisfies the following property: For every process id k and every positive integer s there exists $t \geq s$ such that $k = Enum(t)$. That is, every process id k appears infinitely often. We do not care how the function $Enum()$ is implemented. (A canonical example to scan an infinite array $b[0], b[1], ...$ of bits, is to first read $b[1]$; then $b[1], b[2]$; then $b[1], b[2], b[3]$, etc.)

2.2 Properties of algorithms

We define several algorithm properties considered in the paper.

Adaptive algorithms: An algorithm is *adaptive* with respect to time complexity if the time complexity of processes' operations is bounded by a function of the actual concurrency. (Several time complexity measures are defined in Section 2.3.) Adaptivity can also be defined relative to any complexity measure, not just time. Throughout the paper, by *adaptive* we mean adaptive with respect to time complexity. (In [MT93], the term *contention sensitive* was used but later, the term *adaptive* become commonly used.)

Symmetric algorithms: An algorithm is *symmetric* if the only way for distinguishing processes is by comparing (unique) process identifiers. In such algorithms, process id's can only be written, read, and compared. In particular, identifiers cannot be used to index shared registers. Variants of symmetric algorithms can be defined depending on how much information can be derived from the comparison of two unequal identifiers. In this paper, we assume that id's can only be compared for equality.

In particular, there is no total order on process id's in symmetric algorithms. Styer and Peterson discuss two types of symmetry [SP89]. Our notion corresponds to their stronger restriction, "symmetry with equality". Some of our asymmetric algorithms, presented later, satisfy their weaker symmetry restriction, "symmetry with arbitrary comparisons" in that they depend upon a total order of the process id's, but do not (for example) use id's to index arrays.

Anonymous algorithms: An algorithm is *anonymous* if the processes are initially completely identical [Ang80, AGM98, ASW88]. In particular, the processes do not initially have identifiers and they execute the same code, and hence there is no way at all for distinguishing processes. In Section 6.2, we study the *naming* problem in this very weak model, in which processes are initially identical and utilize shared objects to acquire unique names.

Wait-free algorithms: Several progress conditions have been proposed in the literature for concurrent algorithms which do not use locks. The most studied condition is wait-freedom [Her91]. An algorithm is *wait-free* if it guarantees that *every* process will always be able to complete its pending operations in a finite number of its own steps regardless of the execution speed of other processes.

Advantages of using wait-free algorithms are that they are not subject to deadlocks, they are resilient to process crashes, and they do not suffer significant performance degradation from scheduling preemption, page faults, or cache misses. Such algorithms are still not used in many practical applications as such algorithms are often complex and memory consuming.

2.3 Complexity Measures

The time efficiency of a synchronous algorithm is often of crucial importance. However, it is difficult to find the appropriate definition of time complexity, for systems where nothing is assumed about the speed of the processes. There are two approaches for measuring time: one is to count steps and the other is to assume that a step takes at most one time unit. We describe both approaches below.

2.3.1 Counting Steps

Contention between processes should be rare in well designed systems. Hence, a desirable property of a solution is that, in the absence of contention, a process can complete its operation in a few steps. This leads us to our first definition of time complexity.

Contention-free time complexity: The maximum number of times (i.e., steps) a process may need to access the shared memory in order to complete its operation in the absence of contention. In the context of the mutual exclusion problem, we count the maximum number of times (i.e., steps) a process may need to access the shared memory in its entry code in order to enter a critical section in the absence of contention, plus the maximum number of accesses to the shared memory in its exit code.

It is also important to try (when possible) to minimize the time it takes to enter the critical section when there is contention. To take into account the interactions among processes, two complementary definitions are proposed.

Process time complexity: The maximum number of times a process may need to access the shared memory in order to complete its operation. In the context of the mutual exclusion problem, we count the maximum number of times (i.e., steps) a process that gets to enter its critical section may need to access the shared memory in its entry code and exit code, since the last *time some process released its critical section.*

The above definition is also called *step complexity.* This definition does not always capture the actual running time of an algorithm, for if one process is in a loop waiting for another to complete some action, then the faster it goes, the more times it will go around the loop. But making the first process go faster does not necessarily degrade the total system performance, although it certainly increases the amount of "wasted" effort by the first process. This leads to the following alternative definition.

2.3.2 Counting Time Units

Assume that every step of every process takes some amount of time in (0,1]. That is, there is an upper bound 1 for the time to complete each step, but no lower bound. Thus, for example, if during some period of time where two processes are active, one process takes 100 steps while the other takes 5 steps, then the total time that has elapsed is at most 5 time units. We measure the first time unit to the last point at which each of processes that has begun executing the algorithm takes at most one step. Put another way, time 1 ends and time 2 begins, exactly when some process starts taking its second step. From any later point in time t, we measure one additional time unit to the first point at which all the processes that were active at time t, and have not crashed or completed their operations, have taken at least one step.

System response time: The longest time it takes some process to complete its operation, assuming there is an upper bound of one time unit for each step time and no lower bound. In the context of the mutual exclusion problem, we count the longest time interval during which some process is in its entry code and no process is in its critical section, assuming there is an upper bound of one time unit for each step in the entry or exit code and no lower bound.

The system response time is bounded from below by the contention-free time complexity. The notion of system response time is similar to the notion of round complexity (which is sometimes also called "big step" complexity) when defined in the context of asynchronous systems.

2.3.3 Space Complexity

Finally, it is also desirable to minimize the space complexity, which is defined as follows:

Space complexity: The number and size of the shared memory objects used in the solution.

3 Atomic registers: some participation is required

This section demonstrates that requiring a minimum number of processes to participate is a powerful enabling assumption: the problems we study are solvable with a small number of registers. However, reducing the number of registers does not mean that it is also possible to achieve a finite shared state space. We begin by showing that any solution to election in this model requires infinite shared space. Then we show that election can be solved either by using an infinite number of binary registers, or a single register of infinite size.

3.1 A lower bound for election when the concurrency is $c > 1$

Even when the number of processes that are required to participate equal to the concurrency, any solution to election for an unbounded number of processes must use infinite shared space. This holds even if processes do not need to learn the identity of the leader as part of the election algorithm.

Theorem 3.1 *There is no solution to election with known bounded concurrency $c > 1$ and with $[c, \infty]$-participation using finite shared memory (finitely many registers of finite size).*

Proof: Assume to the contrary that there is such an algorithm using finite space. Consider the solo run $solo(p)$ of each process p from the initial state, and define $write(p)$ to be the sequence of states of the shared memory in *solo*(p). (Because $c > 1$, *solo*(p) may be infinite, as process p waits for the arrival of additional processes.) For each process p, let *repeat*(p) be the first state of the shared memory that repeats infinitely often in *solo*(p). (Recall that the algorithm uses finite space by assumption. Also, if *write*(p) is finite, define *repeat*(p) to be the state after the last write in *write*(p).)

Let $\beta(p)$ be the finite prefix of *write*(p) that precedes the first instance of *repeat*(p). For each such $\beta(p)$, we construct a *signature* subsequence, *sign*(p), by removing repetitive states and the intervening states, as follows: Let $\beta(p) = x_0, x_1, x_2 ...$, where the x_i are the successive states of the shared memory in $\beta(p)$. Suppose j is the index of the first state that repeats in $\beta(p)$ and k is the index of the last occurrence of state x_j in $\beta(p)$. Remove the subsequence $x_{j+1}, ..., x_k$ from $\beta(p)$. The resulting sequence $x_0, ..., x_j, x_{k+1}, ...$ is a subsequence of $\beta(b)$ with strictly fewer repeating states. Repeat this step (finitely many times) until no state repeats. The resulting sequence is (the signature) *sign*(p).

Since there are only finitely many states of the shared memory, there exists a single state s and a single sequence γ such that $s =$ *repeat*(p) and $\gamma =$ *sign*(p) for infinitely many processes, p. Let $C = \{p_1, ..., p_c\}$ be any subset of c of these processes. We show that the solo runs of processes in C are *compatible*: There exists a single infinite run *elect*(C) of the processes in C in which each process p_i takes the same steps as in *solo*(p_i).

The run *elect*(C) is constructed as follows: Let $\gamma = y_0 y_1, ..., y_k$. Run p_1 until its first write to the shared memory that changes the shared memory state. Do the same for p_2 through p_c. Now each process p_i is about to execute its first write that actually changes the value of shared memory, from y_0 to y_1. Run process p_1 until it arrives at the last state in $\beta(p_1)$ that leaves the memory in state y_1. (Note that y_0 and y_1 differ by the value of exactly one word of the shared memory, which is exactly the word that each of p_2 through p_c are preparing to change.) Repeat for p_2 through p_c, so that each process p_i in $\{p_1, ..., p_c\}$ has executed the steps of *solo*(p_i) up to the last occurrence of y_1 in $\beta(p)$. Now repeat this loop $k - 1$ more times, for each remaining state in γ, so that in the j-th iteration, each process p_i in $\{p_1, ..., p_c\}$ has executed the steps of *solo*(p_i) up to the last occurrence of y_j in $\beta(p)$.

The resulting finite run leaves the shared memory in state s, and each of the c processes p_i has taken all the steps in $\beta(p_i)$. Since the state s repeats infinitely often in the remainder of each of the solo runs *solo*(p_i), the run can be extended, using a round-robin schedule, appending a finite sequence of steps of each process in turn, always returning the memory to state s. The resulting run is *elect*(C).

To conclude the proof, note that since *elect*(C) has concurrency at most c and participation c, one of the participants, p_i, must be elected. Similarly, for a set of processes $C' = \{p'_1, ..., p'_c\}$ disjoint from C, in the run *elect*(C') some process $p_{i'}$ is elected. Finally, including both p_i and $p_{i'}$ in a subset C'' of $C \cup C'$ of size c results in a run *elect*(C'') in which both p_i and $p_{i'}$ are elected. This contradicts the assumption that the algorithm is correct. ∎

3.2 Algorithms for election and consensus for concurrency c

The previous theorem shows that election for an unbounded number of processes requires unbounded shared memory. In this section, we present algorithms for the required participation case that bound either the number of shared words or their size. We first study the scenario in which the concurrency is equal to participation (concurrency c and participation $[c, \infty]$). We show that election can be solved either by (1) using an infinite number of atomic bits, or (2) using a single register of infinite size. We also present two simple symmetric algorithms.

Theorem 3.2 *For any known bounded concurrency c, with $[c, \infty]$-participation, there are non-adaptive asymmetric solutions to election (and consensus) using an infinite number of atomic bits.*

Proof: We identify each process with a unique natural number, and assume 0 is not assigned to any process. The algorithm uses an infinite array $b[0], b[1], ...$ of bits, which are initially 0.

The first step of process i is to read $b[0]$. If the value seen is 1, process i knows that a leader has already been elected and terminates. Otherwise, process i sets $b[i]$ to 1. Then, process i scans the array until it notices that c other bits (other than $b[0]$) are set. In order not to miss any bit that is set, it uses the function *Enum*() (as defined in Section 2.1) to scan the array according to an infinitely repeating enumeration. Once a process notices that c bits are set, it sets $b[0]$ to 1, chooses the process with the smallest id among the c that are set to 1 as the leader, and terminates. (By scanning the bits after reading $b[0]$ as 1, the other processes can also learn the id of the leader.)

Only processes which read $b[0] = 0$ will write any bits, and no such process terminates until it sees c other bits set. Hence, the first c processes do not terminate until all of them read $b[0] = 0$ and set their bits. The first one to terminate sets $b[0]$ to its final value of 1. Only then can the $c + 1$'st process become active. Hence, exactly $c + 1$ bits will eventually be set.

This solution for election can be trivially modified, using one other bit, $b[-1]$, to solve consensus. Each process i (among the first c) which reads $b[0] = 0$ sets $b[-1]$ to its input value before setting $b[i] = 1$ and continuing with the rest of the algorithm. Before terminating, each process reads and returns the value of $b[-1]$. The consensus value will be the input of the c-th process, to write to $b[-1]$. ∎

A symmetric election algorithm is presented in [SP89], for n processes (with concurrency level n), which use only three atomic registers. Below we present three election algorithms: (1) a symmetric algorithm for concurrency 2 using one register, (2) a symmetric algorithm, using an idea from [AGM98], for concurrency c using two registers, and finally, (3) an asymmetric algorithm for concurrency c using only a single register.

Theorem 3.3 *For known bounded concurrency $c = 2$ with $[2, \infty]$-participation, there are adaptive symmetric solutions to election and consensus using one atomic register.*

Proof: The algorithm in Figure 1 is very simple. The register value contains two fields, *Leader* and *Marked*, respectively, where *Leader* is a process id (or 0) and *Marked* is boolean. The statement **await** *condition* is used as an abbreviation for **while** $\neg$*condition* **do** *skip*, and hence, may involve many accesses to the shared memory.

In this very simple algorithm the second process to write to the shared register is elected. The first process to write spins until the second process writes, then marks the second process as the winner. The concurrency bound guarantees no other process will interfere.

This election algorithm can be easily converted into a consensus algorithm by appending the input value (0 or 1) to each process id. The input value of the leader is the consensus value. ∎

```
Process i's program
Shared:
        (Leader, Marked): (Process id, boolean) initially $(0, 0)$
Local:
        local_leader: Process id

1   if $Marked = 0$ then
2       $(Leader, Marked) := (i, 0)$
3       await $(Leader \neq i)$ or $(Marked = 1)$
4       $local\_leader := Leader$
5       $(Leader, Marked) := (local\_leader, 1)$
6   fi
7   return(Leader)
```

Figure 1: Symmetric election for concurrency 2

The next algorithm, solving asymmetric election for concurrency $c \geq 2$, is similar to the previous, in that the last participant to write the *Leader* field is then *Marked* as the leader. A second register, *Union*, is used by the first c participants to ensure that they have all finished writing their id into *Leader*. (Note, that in doing so, they learn each other's id's.) Because c is also the number of required participants, each of the first c participants can spin on the *Union* register until the number of id's in it is c.

Theorem 3.4 *For any known bounded concurrency c with $[c, \infty]$-participation, there are adaptive symmetric solutions to election and consensus using two atomic registers.*

Proof: The algorithm in Figure 2 uses a key idea used previously in a symmetric algorithm for a general function evaluation problem, due to Attiya et al. [AGM98]. As in the Attiya et al. algorithm, the first register, *Union*, holds a set of at most c processes id's. Processes engage in a phase of reading the current set of id's, comparing it to the set they have already seen, and writing when they know an id not reflected in the set read. This phase ends when a process reads c different id's. (In the Attiya et al. algorithm, this register holds a set of inputs to a collectively-computed function. This phase ends when a process observes enough inputs to uniquely determine the function value, which is written in the second register, providing the termination condition and output value for all participating processes.)

In the algorithm in Figure 2, the second register is used both to regulate access to the *Union* register, and to break symmetry among the set of process id's. This is necessary because we only allow equality comparisons. It contains two fields: a process id (in the *Leader* field) and a bit (the *Marked* field). Processes access the *Union* register only if they read 0 from the *Marked* field. In addition, no process terminates until it sets the *Marked* field to 1. The concurrency bound guarantees that the $c+1$'st process cannot take a step until one of the first c processes terminate, by which time the *Marked* field is set. Hence, only the first c processes access the *Union* register. We prove below that one of these processes must eventually see c different process id's (and go on to set *Mark*). Hence, the leader will be the c-th process to write to *Leader*.

The proof begins by arguing that all the processes that terminate elect exactly the same participant as a leader. Note first that only participating process id's can appear

```
Process $i$'s program
Shared:
        $(Leader, Marked)$: (Process id, boolean), initially $(0, 0)$
        $Union$: set of at most $c$ process id's, initially $\emptyset$
Local:
        $local\_leader$: Process id
        $local\_union1$: set of at most $c$ process id's
        $local\_union2$: set of at most $c$ process id's, initially $\{i\}$

1   if $Marked = 0$ then $(Leader, Marked) := (i, 0)$ fi
2   $local\_union1 := Union$
3   while $(|local\_union1| < c) \wedge (Marked = 0)$ do
4       if $\neg(local\_union2 \subseteq local\_union1)$ then
5           $Union := local\_union1 \cup local\_union2$
6       fi
7       $local\_union2 := local\_union1 \cup local\_union2$
8       $local\_union1 := Union$
9   od
10  $local\_leader := Leader$
11  $(Leader, Marked) := (local\_leader, 1)$
12  return$(Leader)$
```

Figure 2: Symmetric election for concurrency c

in the shared *Union* register. A process sets *Marked* from 0 to 1 and terminates only after it determines that c processes are already participating. Thus, the first c processes to participate will first see that *Marked* is not set, and no later process can participate until one of these terminates, after setting *Marked*. Hence, only the first c processes will write their id's into *Leader*. Moreover, no process sets *Marked* until the last of these c processes have written to *Leader*. Hence, any processes that terminate will elect the last process to write its id to *Leader*.

To prove termination, consider the finite partial order defined by the subset relation on the subsets of the set of names C of the first c processes. We can consider each of the first c processes as moving from one point to another in this partial order as their *local_union2* variables are updated. (Several processes may move to the same point.) Because these variables are only updated by union with their previous value, processes can only move upward in this order. (Call the set of id's in *local_union2* the set that the associated process currently *knows about*.) As long as *Marked* is not set, we argue that the only stable state (i.e., states in which the value of *Union* will not change) is one in which every process knows about every other. (That is, $|local\_union2| = c$ for each of the first c processes.)

Consider any other stable state, and assume that the marking of the partial order is stable in a run extending it, in which each of the first c processes either takes an infinite number of steps or terminates. This state contains minimal elements: processes with $|local\_union2| < c$ and such that no other process knows only about a strict subset of $local\_union2$. Once a minimal process writes its value of $local\_union2$ to *Union*, no strict subset of that set can be written. Each such process can write at most once in the run extending this state, as it writes only if it learns of a new process, which by assumption it cannot. So in its (infinitely many) subsequent reads, it must see exactly the set it currently knows. But eventually a process that knows a different id will write to the *Union* register and then be read by a minimal process. This is a contradiction.

It follows that at some point a process will learn that c processes are participating (i.e., $Union = C$ and $|Union| = c$) and set *Marked* to 1, allowing the rest of the processing to terminate.

Establishing adaptivity means bounding the time complexity of each participating process by a function of c. For a subset K of C, define $W(K)$ as the maximum number of times any single process can write K to *Union*. (Note that only processes in K can write K to *Union*.) Each write must be preceded by the read of a strictly smaller set. Hence, for any $p \in C$, we have $W(\{p\}) = 1$, and $W(K) \leq \Sigma_{K' \subset K} |K'| \cdot W(K')$. Hence the number of distinct writes to *Union* is bounded above by $\Sigma_{K \subset C} |K| \cdot W(K)$, a function of c.

This completes the correctness proof of the election algorithm.

As above, this election algorithm can be easily converted into a consensus algorithm, by adding the input value (0 or 1) to each process id. ∎

Note that the termination argument above holds even if some processes stop updating the *Union* register, as long as each active process knows about everyone. This observation is important to the next algorithm, an asymmetric solution to election (and consensus) using only a single atomic register. Asymmetric algorithms allow comparisons between process id's (we assume a total order). However, since the identities of the participating processes are not initially known to all processes, the trivial solution where all processes choose process 1 as a leader is excluded.

Theorem 3.5 *For any known bounded concurrency c and $[c, \infty]$-participation there is an adaptive asymmetric solution to election and consensus using one atomic register.*

Proof: The one-register asymmetric algorithm in Figure 3 can be understood as an adaptation of the previous, two-register algorithm for the symmetric case (Theorem 3.4). As in an asymmetric one-register algorithm due to Attiya et al. [AGM98], it modifies the symmetric algorithm by adding a leader-lead handshake phase. (The algorithm here has slightly different termination conditions than the function evaluation algorithm, to prevent late processes from running concurrently with the first c active processes. We also prove adaptivity of the algorithm.)

As before, each of the first c processes (denoted by the set C) repeatedly updates the shared *Union* register with any new information it obtains. The process elected as leader is the process with the minimum id among the first c processes to participate in the algorithm. Termination detection among the first c processes is accomplished by establishing an explicit handshake between the leader and each of the other $c - 1$ initial processes.

In outline, since the size of the set in *Union* is never more than c, once a process, say p sees that *Union* has size at least c, the process p knows the identity of the leader, which is the minimum element in this set. If p is not in the set (is not one of the first c participants) it simply terminates. Otherwise, p performs a handshake with the leader, and terminates once it observes that all the $c-1$ handshakes have completed. The handshake is carried out using additional fields in the register: a counter, *Sig*, by which the leader signals to the initiating process with rank *Sig* in *Union*, and a bit, *Ack*, by which this process acknowledges. That is, process p waits until the leader sets *Sig* to p's rank among the c initial processes, and then sets *Ack* to 0 to acknowledge the signal. (Notice that we are counting rank starting

```
Process $i$'s program
Shared:  $(Union, Sig, Ack)$: $Union$ a set of at most $c$ process id's,
             $0 \le Sig \le c$, $Ack \in \{0,1\}$, initially $(\emptyset, 0, 0)$
Local:   $local\_union1$: set of at most $c$ process id's
         $local\_union2$: set of at most $c$ process id's, initially $\{i\}$
         $myrank, local\_sig1, local\_sig2$: integers between 0 and $c$, initially 0
         $local\_ack1$, $acked$: boolean, initially 0
         $leader$: process id

1   $(local\_union1, local\_sig1, local\_ack1) := (Union, Sig, Ack)$
2   while $|local\_union1| < c$ do
3       if $\neg(local\_union2 \subseteq local\_union1)$ then
4           $(Union, Sig, Ack) := (local\_union1 \cup local\_union2, 0, 0)$
5       fi
6       $local\_union2 := local\_union1 \cup local\_union2$
7       $(local\_union1, local\_sig1, local\_ack1) := (Union, Sig, Ack)$
8   od
9   $local\_union2 := local\_union1$
10  $leader := \min\{q : q \in local\_union1\}$
11  if $i \notin local\_union1$ then
12      return(leader)
13  elseif $i \neq leader$ then
14      $myrank := |\{h \in local\_union1 : h < i\}|$
15      while $local\_sig1 < c$ do
16          if $(|local\_union1| < c) \wedge (acked = 0)$ then /* register is stale, haven't acked */
                $(Union, Sig, Ack) := (local\_union2, 0, 0)$
17          elseif $(local\_sig1 = myrank) \wedge (local\_ack1 = 1)$ then
18              $(Union, Sig, Ack) := (local\_union2, myrank, 0)$ /* acknowledge signal */
19              $acked := 1$
20          fi
21          $(local\_union1, local\_sig1, local\_ack1) := (Union, Sig, Ack)$
22      od
23      return($leader$)
24  else                                          /* $|local\_union2| = c$, $i = leader$ */
25      while $local\_sig1 < c$ do
26          if $(|local\_union1| < c) \vee (local\_sig1 < local\_sig2)$ then
27              $(Union, Sig, Ack) := (local\_union2, local\_sig2, local\_ack2)$/* register is stale */
28          elseif $local\_ack1 = 0$ then                  /* signal acknowledged */
29              $local\_sig2 := local\_sig2 + 1$
30              $(Union, Sig, Ack) := (local\_union2, local\_sig2, 1)$ /* send new signal */
31          fi
32          $(local\_union1, local\_sig1, local\_ack1) := (Union, Sig, Ack)$
33      od
34      return($i$)
35  fi
```

Figure 3: (Asymmetric) election for concurrency c

at 0.) Except for acknowledging the signal, p stops participating in the updates to $Union$, but waits to terminate until the final signal value of c is posted. (This precludes any but the first c active processes from participating until the register value has the stable value of $(C, c, 1)$. Moreover, at the point that process p acknowledges the signal, it knows that the leader knows all c initial id's, and will eventually refresh $Union$ if necessary.) The leader itself initiates each handshake in order once it knows the id's of all c initial processes. Slow processes may overwrite handshakes in progress, which the leader (or higher-ranking processes) will detect and refresh. We notice that signal value c means the handshaking is complete.

As in the previous algorithm, no process exits the while loop in line 2 until c processes have entered the loop. We argue below that none of the initial processes terminates until the shared register contains the final and stable value $(C, c, 1)$. It follows that all processes that terminate elect the same leader.

To argue that every process terminates, the main difficulty is arguing that the initial c processes terminate. Note that no process stops updating $Union$ until: (1) it knows the id's of c participating processes (the initial while loop in line 2), and (2), until the update in line 16 is skipped once the leader's signal is acknowledged (which intuitively means that the process knows that the leader knows the id's of c participating processes). As in the argument in the preceding algorithm, this implies that all processes in C will eventually learn the id's in C.

By induction on the rank $r \geq 1$ of process p in C, process p eventually reads its signal from the leader, the leader eventually reads its acknowledgment, and (using the explicit *acked* flag to avoid unnecessary refreshes of the register) process p never executes another write operation. Hence, the leader will eventually set the register to $(C, c, 1)$ and terminate. The other initial participants (who have stopped writing but are still reading) will eventually read $Sig = c$ and terminate, and as we argued above, the later processes will all see c different values in $Union$ and terminate immediately.

As in the previous algorithm, establishing adaptivity means bounding the time complexity of each participating process by a function of c. The proof of Theorem 3.4 defined a recurrence based on the finite partial order of the subsets of the set C of the first c active processes. To prove the algorithm in Figure 3 adaptive, extend the subset partial order with the $2c - 1$ handshake values in the linear order of the handshake:

$$C < (C, 1, 1) < (C, 1, 0) < (C, 2, 1) < (C, 2, 0) < \cdots < (C, c, 1)$$

The same recurrence argument as in the proof of Theorem 3.4 establishes adaptivity by bounding the number of times any value can be written. ∎

4 Atomic registers: participation is not required

As mentioned above, any problem solution in a model in which participation is not required, is also correct when participation is required, and hence such solutions may be more difficult to construct. The main new result of this section is an adaptive starvation-free mutual exclusion algorithm for unbounded concurrency.

4.1 Consensus, election and mutual exclusion for concurrency c

We next consider the number and size of registers required to solve consensus, election and mutual exclusion in a model in which participation is not required (i.e., $[1, \infty]$-participation). Earlier work on these problems for known bounded concurrency establishes some bounds:

Theorem 4.1 *For concurrency level c, the number of atomic registers that are:*
(1) necessary [BL93] *and sufficient* [SP89] *for solving deadlock-free mutual exclusion, is c;*
(2) necessary and sufficient for solving election, is $\log c + 1$ [SP89]*;*
(3) sufficient for solving consensus, is $\log c + 1$ [SP89].

Proof:

(1) Any deadlock-free mutual exclusion algorithm for n processes must use at least n shared registers [BL93]: this bound applies immediately to concurrency.

There is a symmetric deadlock-free mutual exclusion algorithm for n processes which uses only n shared registers [SP89]. This algorithm works without modification for an unbounded number of processes, provided that the concurrency level is no more than n.

(2) Any election algorithm for n processes must use at least $\log n + 1$ shared registers [SP89]: as above, this bound applies immediately to concurrency.

There is a symmetric election algorithm for n processes which uses only $\log n + 1$ shared registers [SP89]: it also works without modification for an unbounded number of processes, provided that the concurrency level is no more than n.

(3) The symmetric election algorithm in [SP89] can be easily modified to solve consensus with the same number of registers: Each process appends its vote (0 or 1) to its id. Processes choose the leader's vote as the consensus value. ∎

4.2 Mutual exclusion algorithms for unbounded concurrency

Theorem 4.1 implies that when the concurrency level is not known bounded, an infinite number of registers are necessary for solving the election and mutual exclusion problems. But do an infinite number of registers suffice for solving mutual exclusion for unbounded concurrency, when participation is not required?

We point out that, based on a result of Yang and Anderson [YA96], if we restrict the number of processes that can try to write the same register at the same time, the answer is no. This observation follows from one of the results in [YA96] which implies an $\Omega(\log_w n)$ lower bound on the contention-free time complexity of any algorithm that uses atomic registers, where w is the maximum number of processes that may simultaneously write the same register.

For the general case, where there are no restrictions on the number of processes that can try to write the same register at the same time, we present two algorithms that answer this question affirmatively. The first is an adaptive and symmetric mutual exclusion algorithm using infinitely many infinite-sized registers. The second is neither adaptive nor symmetric, but uses only (infinitely many) bits.

Theorem 4.2 *There is an adaptive symmetric solution to starvation-free mutual exclusion for unbounded concurrency (and, hence, to election and consensus) using an infinite number of registers.*

These problems can all be solved by simple adaptations to the deadlock-free mutual exclusion algorithm presented in Figure 6 below. This algorithm has the following interesting properties:

1. It works for unbounded concurrency.
2. In the absence of contention, only *eight* accesses to the shared memory are needed (*seven* in the entry code and *one* in the exit code).
3. It is adaptive – its time complexity (even in the worst case, measured from the exit of the critical section by a process to the next entrance by any process) is a function of the actual number of contending processes.
4. It is symmetric – identifiers are only written, read and compared for equality, but are not ordered or used to index shared registers.

Except for the non-adaptive algorithm in Section 4.2.1, we know of no mutual exclusion algorithm (using atomic registers) satisfying the first property. The algorithm is defined formally in Figure 6, and is built out of simple building blocks called *splitters* which are introduced next.

4.2.1 Splitters

In [Lam87], Lamport presented a mutual exclusion algorithm which provides fast access in the absence of contention. As emphasized by Moir and Anderson [MA95], his algorithm makes use of a shared object called a *splitter*. This object is shown in Figure 4. We say that a process *exits* the splitter object, when an operation that is invoked by that process on the splitter returns. Each process that invokes the splitter moves either *down*, *right* or *wins*. In any execution, define the *latecomers* to be the processes that invoke the splitter after the first process exits it. Let n be the number of *early* processes that invoke the splitter before this first process exits it. Properties of Lamport's splitter are listed below.

Proposition 4.3 *In any execution of Lamport's splitter, the following properties hold:*

1. *At most $n-1$ early processes move right,*
2. *at most $n-1$ processes move down, or at least one latecomer moves right.*
3. *at most one process wins, and if $n=1$ then exactly one process wins,*
4. *the latecomers all move right, and*
5. *the splitter is wait-free.*

Proof: To see, for example, why the third property holds, assume to the contrary that two processes i and j both *win*. Assume without loss of generality that process i tests the value of x at statement 4 after j does. This implies that x is not written by any process between i's assignment $x := i$ in statement 1 and i's read of x in statement 4. Thus, j's read of x in statement 4 preceded i's assignment in statement 1, which in turn implies that j assigned 1 to y in statement 3 before i's read of y in statement 2. Thus, i must have read $y = 1$ at statement 2 and then "moved right", a contradiction. Similar arguments establish the other properties. ∎

```
Process $i$'s program
Shared:
$x$: integer (the initial value is immaterial)
$y$: boolean, initially 0

1 $x := i$
2 if $y = 1$ then goto right fi
3 $y := 1$
4 if $x \neq i$ then goto down
5 else goto win fi
```

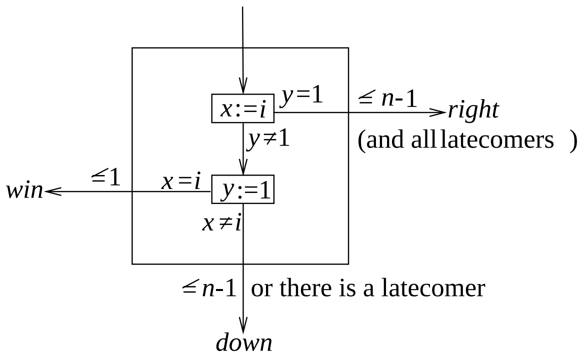

Figure 4: *Lamport's splitter.*

Next, we modify Lamport's splitter, exchanging wait-freedom for starvation-freedom, but obtaining a new safety property.

Lemma 4.4 *The new splitter in Figure 5 satisfies the first four properties of Lamport's splitter (as listed in Proposition 4.3), is starvation-free, and satisfies the following additional property: If a process wins then nobody moves down.*

Proof: Starvation-freedom follows because Lamport's splitter is wait-free and the conditions satisfying the *await* statement are stable (i.e. once b and z are set to 1, they remain 1). The second property of Lemma 4.3 assures that some process must exit Lamport's splitter to the *right* or by *winning*, and perform the assignment enabling all *await* statements to terminate. To see why the new safety property holds, observe that:

1. a process that reaches the await statement in line 4, (exiting Lamport's splitter going down) can move *down* in the new splitter only if b is set to 1 *before* z is set to 1, and

2. a process can *win* in the new splitter only if b is set to 1 *after* z is set to 1.

Thus, if a process *wins* in the new splitter, every process that reaches the await statement in line 4 moves *right* in the new splitter. The winning process is also the (unique) process to *win* in Lamport's splitter, and so nobody moves *down.*

The other four safety properties of Lamport's splitter also hold for our new splitter. Notice first that any process that is an *early* process in the embedded Lamport's splitter is also an *early process* in the new splitter, and any process that is a *latecomer* in our new splitter is a *latecomer* in the embedded Lamport's splitter. The third and fourth safety properties follow immediately from the corresponding conditions in Lemma 4.3. To see the first safety property holds, observe that one of the processes that are *early* in Lamport's splitter exits it as a *winner* or *down.* In the first case, we are done, so assume no process *wins* in Lamport's splitter. Then the processes exiting Lamport's splitter *down* will continue and exit the new splitter going *down.*

Finally, we prove that the second safety property holds. If some process moves *right* in the new splitter (either an *early* process or a *latecomer*) then we are done. If no process moves *right* in the new splitter, then also no process moves *right* in Lamport's splitter, and by the second safety property in Lemma 4.3, exactly one process (*early* in both splitters) *wins* in Lamport's splitter, and then in the new splitter. This is so, because in this case, b is never set to 1. ∎

```
Process $i$'s program
Shared:
$x$: integer (the initial value is immaterial)
$b, y, z$: boolean, initially 0

$x := i$
if $y = 1$ then $b := 1$ goto right fi
$y := 1$
if $x \neq i$ then await $((b = 1)$ or $(z = 1))$
                   if $z = 1$ then goto right
                   else goto down fi
else   $z := 1$
       if $b = 0$ then goto win
       else goto down fi fi
```

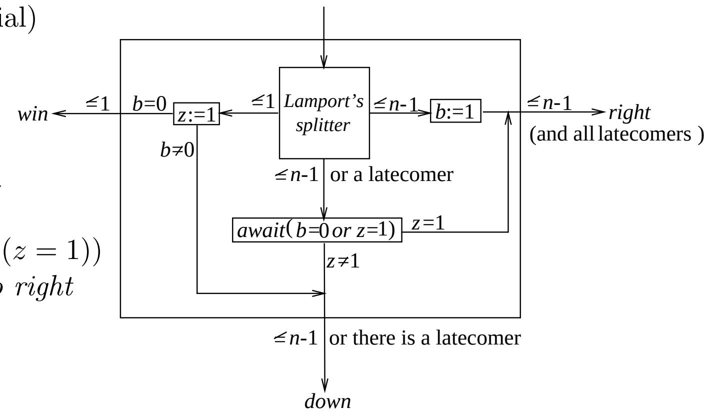

Figure 5: The code of the *new splitter*.

4.2.2 An adaptive mutual exclusion algorithm

Using the properties of our new splitter, it is possible to solve mutual exclusion by interconnecting a collection of splitters in an (infinite) chain, so that processes that move *down* enter the next splitter in the chain, as is illustrated in Figure 6.

Lemma 4.5 *The algorithm in Figure 6 is an adaptive, deadlock-free mutual exclusion algorithm for unbounded concurrency.*

Proof: In this solution, the processes compete in levels, where each level is implemented as a separate new splitter. Each level either has a *winner*, or eliminates at least one competing process (those that move *right*). Since all the *latecomers* move *right* at the first level, only finitely many processes can move *down* to subsequent levels. Hence, within finitely many levels, there is either a *winner* or a single process moves down–but this process will *win* this final level. The *winner* enters its critical section, and in its exit code it publishes the index of the next empty level, so that each process can join (or rejoin) the competition starting from that level.

By the additional property of the new splitter, namely that "if a process wins, then no process moves down", when a process *wins* in a level, no process is active in any splitter with greater index. Every other active process will eventually move to the *right* in a splitter with equal or lesser index, and wait for the global pointer *global_level* to be updated by the *winner*.

Measuring time complexity, exiting the critical section signals awaiting processes to proceed to a splitter at a new level. If n is the number of *early* processes at that splitter, then the next *winner* accesses $n-1$ additional splitters (i.e., a total of at most $n+1$ splitters) before entering the critical section. This takes at most $O(n)$ time. Hence, the algorithm is adaptive.

Examining the details of the code, in the absence of contention, only seven accesses to shared atomic registers are needed before entering the critical section. ∎

```
Process $i$'s program
**Shared**:
    $global\_level$: integer, initially 0
    $x[0..\infty]$: array of integers (the initial values are immaterial)
    $b[0..\infty]$, $y[0..\infty]$, $z[0..\infty]$: array of boolean, initially all 0
**Local**:
    $my\_level$: integer (the initial value is immaterial)

    $my\_level := global\_level$
  $start$: $x[my\_level] := i$
    **if** $y[my\_level]$ **then** $b[my\_level] := 1$ $goto\ right$ **fi**
    $y[my\_level] := 1$
    **if** $x[my\_level] \neq i$ **then await** $((b[my\_level] = 1)\, or\, (z[my\_level] = 1))$
            **if** $z[my\_level] = 1$ **then** $goto\ right$
            **else** $goto\ down$ **fi**
    **else** $z[my\_level] := 1$
      **if** $b[my\_level] = 0$ **then** $goto\ win$
      **else** $goto\ down$ **fi fi**

  $right$: **await** $my\_level < global\_level$
    $my\_level := global\_level$
    **goto** $start$

  $down$: $my\_level := my\_level + 1$
    **goto** $start$

  $win$: $critical\ section$
    $global\_level := my\_level + 1$
```

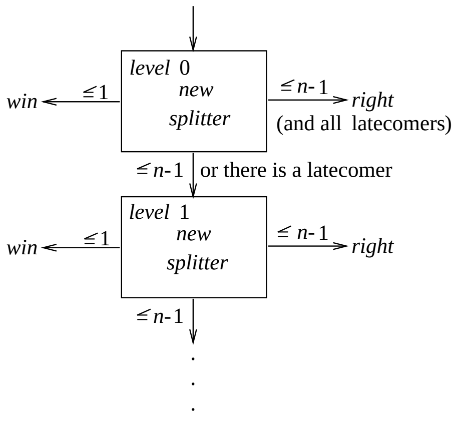

Figure 6: Adaptive deadlock-free mutual exclusion for unbounded concurrency.

4.2.3 Proof of Theorem 4.2

The main theorem of this section, Theorem 4.2, is an immediate consequence of the following.

Lemma 4.6 *The algorithm in Figure 7 is an adaptive, starvation-free mutual exclusion algorithm for unbounded concurrency.*

Proof: The adaptive deadlock-free algorithm of Figure 6 is easily modified using standard "helping" techniques to satisfy starvation-freedom. Because the number of processes is unbounded, a process helps others in the order given by the function $Enum()$, in which every process id appears infinitely often. The details are in Figure 7. Process i sets a flag $try[i]$ when it leaves the remainder section, and before leaving the critical section, a process examines the flag of the process, where cs_counter is, and, if it is trying, lets it enter its critical section.

Starvation-freedom follows. Adaptivity follows just as in the deadlock-free algorithm: recall that in defining adaptivity for mutual exclusion algorithms, we measure the maximum

```
Process $i$'s program
**Shared**:
    $global\_level$, $cs\_counter$: integer, initially both 0
    $x[0..\infty]$: array of integers (the initial values are immaterial)
    $b[0..\infty]$, $y[0..\infty]$, $z[0..\infty]$, $try[1..\infty]$: array of booleans, initially all 0
    $winner\_level$: integer (the initial value is immaterial)
**Local**:
    $my\_level$: integer (the initial value is immaterial)

0           $my\_level := global\_level$
**0.1**     $try[i] := 1$
1         $start$: $x[my\_level] := i$
2           **if** $y[my\_level]$ **then** $b[my\_level] := 1$ $goto\ right$ **fi**
3           $y[my\_level] := 1$
4           **if** $x[my\_level] \neq i$ **then await** $((b[my\_level] = 1) or\ (z[my\_level] = 1))$
5                 **if** $z[my\_level] = 1$ **then** $goto\ right$
6                 **else** $goto\ down$ **fi**
7           **else** $z[my\_level] := 1$
8             **if** $b[my\_level] = 0$ **then** $goto\ win$
9             **else** $goto\ down$ **fi fi**
**10**    $right$: **await** $((my\_level < global\_level)$ $or$ $(try[i] = 0))$
**10.1**    **if** $try[i] = 0$ **then goto** $win$ **fi**                 /* process $i$ is helped */
11          $my\_level := global\_level$                      /* no help, keep trying */
12          **goto** $start$
13        $down$: $my\_level := my\_level + 1$
14          **goto** $start$
15        $win$: $critical\ section$
**15.1**    **if** $try[i] = 1$ **then** $winner\_level := my\_level$ **fi**      /* if $i$ was not helped */
**15.2**    $try[i] := 0$                                     /* then post $my\_level$ */
**15.3**    $cs\_counter := cs\_counter + 1$          /* help process $Enum(cs\_counter)$ */
**15.4**    **if** $try[Enum(cs\_counter)] = 1$ **then** $try[Enum(cs\_counter)] := 0$
**16**      **else** $global\_level := winner\_level + 1$ **fi**
```

Figure 7: Adaptive starvation-free mutual exclusion for unbounded concurrency.

time between releasing the critical section, until the critical section is reentered. ∎

Even simpler, standard modifications convert the deadlock-free mutual exclusion algorithm in Figure 6 to solve leader election or consensus: the first process to reach the critical section announces its id or the consensus value in an additional register. This completes the proof of Theorem 4.2.

4.2.4 A non-adaptive mutual exclusion algorithm using atomic bits

The adaptive mutual exclusion algorithm for unbounded concurrency we have just presented uses infinitely many infinite-sized registers. Infinite space is necessary–Alur and Taubenfeld showed that even without contention, $\Omega(\sqrt{n})$ shared bits must be accessed in any mutual exclusion algorithm for n processes [AT96]. Next we show that for a non-adaptive algorithm,

```
Process i's program
Shared:
    RaceOwner[1..∞], RaceOther[1..∞], Win[1..∞], Lose[1..∞]: arrays of booleans, initially all 0
Local:
    index: integer, initially 1

1   $RaceOwner[i] := 1$
2   if $RaceOther[i] = 0$ then $Win[i] := 1$ else $Lose[i] := 1$ fi
3   repeat forever
4       $RaceOther[index] := 1$
5       if $RaceOwner[index] = 1$ then
6           await ($Win[index] = 1$ or $Lose[index] = 1$)
7       fi
8       if $Win[index] = 1$ then $return(index)$ fi
9       $index := index + 1$
10  end repeat
```

Figure 8: (Non-adaptive) leader election for unbounded concurrency using bits

it suffices to use only atomic bits.

Theorem 4.7 *There is a non-adaptive asymmetric solution for election, consensus and starvation-free mutual exclusion for unbounded concurrency using an infinite number of bits.*

Proof: Figure 8 presents a simple algorithm for election. We observe that, in this election algorithm, each participating process eventually also knows the identity of the elected leader. Modification to solve consensus is done as follows. An additional infinite array of single-writer bits, $b[1..\infty]$, is used. Process i's first step is to set $b[i]$ to its input value. Then, process i participates in the election algorithm and finally decides on the input value of the elected leader. As we explain, starvation-free mutual exclusion can be achieved using techniques similar to those in the previous algorithm.

There are four bits associated with each process i, $RaceOwner[i]$, $RaceOther[i]$, $Win[i]$, and $Lose[i]$. The idea is that each process i uses the two bits $RaceOwner[i]$ and $RaceOther[i]$ to race against the other processes. The other two bits are used by the owner process i to signal whether it won or lost. The leader is the minimum i such that $Win[i]$ is set.

Termination: First, a simple argument shows each execution of the wait loop in line 6 eventually terminates: A process waits on line 6 only after it finds in line 5 that $RaceOwner[index] = 1$. This implies that a process with identifier *index* has already executed line 1 in its code. Since there are no faults, this process will also have to finish executing line 2 of its code and, hence, will set either $Win[index]$ or $Lose[index]$ to 1, enabling the process waiting on line 6 to continue. The first process i to perform the read in line 2 will eventually set $Win[i]$. Every other process will either terminate before $index = i$, or see $RaceOwner[i]$ set, wait until $Win[i]$ is set, and terminate.

Election: In the repeat loop, each process j scans the infinite array Win, starting from $Win[1]$, until it finds an entry, denoted *index*, in the Win array, such that $Win[index] = 1$. Once this happens, process j knows that the process with the identifier *index* is the elected leader.

In parallel with scanning the Win array, processes j tries to prevent "latecomers" from being elected. This is done by setting the corresponding bits of the *RaceOther* array to 1. If process j manages to set *RaceOther*$[k]$ to 1, before process k starts, then by the time process k executes line 2 of its code, it will find that *RaceOther*$[k] = 1$ it will lose, and thus will never set its $Win[k]$ bit to 1.

If process k manages to set *RaceOwner*$[k]$ to 1 (on line 1), before process j reads *RaceOwner*$[k]$ (line 5), then process j has to wait until it knows whether it managed to set *RaceOther*$[k]$ to 1, before process k read *RaceOther*$[k]$. This waiting is implemented by letting process j busy-wait (spin) on the $Win[k]$ and *Lose*$[k]$ bits until one of these two bits is set to 1 by process k.

In an infinite run, there is a minimum i such that $Win[i] = 1$. We notice that once $Win[i]$ is set to 1, it will never be set back to 0. Every other process must see *RaceOwner*$[i]$ set, since process i reads *RaceOther*$[i] = 0$ in line 2 after setting *RaceOwner*$[i]$ in line 1. Thus, every process will return i, declaring i as the single elected leader.

A fairly straightforward revision implements deadlock-free mutual exclusion. That is, use infinitely many copies of these binary data structures, one for every entrance to the critical section. Each copy x has a control bit, $Version[x]$. The leader in $Version[x]$ is the x-th process to enter the critical section. The exit code is to set the control bit on the current version, signaling $Version[x+1]$ is now current. To enter the critical section, a process finds the minimum copy with the control bit still open, contends there until it is the leader or (if it isn't) spins until the control bit is set on that copy, then tries again on the next.

To achieve starvation-free mutual exclusion, use the function *Enum*() (as defined in Section 2.1) to find the index to the current copy of the algorithm to determine who to help next. See Figure 9 for details. To implement the *LeaderElect*(x) procedure, for $x \in \{1, .., \infty\}$, process i runs the leader election algorithm of Figure 8 on copy x of the data structure. ∎

5 Test&set bits

A test&set bit is an object that may take the value 0 or 1, and is initially set to 1. It supports two operations: (1) a reset operation: write 0, and (2) a test&set operation: atomically assign 1 and return the old value. We first make the following observation:

Theorem 5.1 *An infinite number of atomic registers are necessary for implementing a test&set bit when concurrency is unknown bounded, and sufficient when concurrency is unbounded.*

Proof: To prove the necessary condition assume to the contrary that such an implementation is possible with a priori known, finite number of atomic registers. This assumption, together with the observation that there exists a trivial deadlock-free mutual exclusion algorithm using a single test&set bit, implies that there is a solution for deadlock-free mutual exclusion with unknown bounded concurrency using a finite number of atomic registers. This contradicts Theorem 4.1. The sufficient condition follows immediately from Theorem 4.7. ∎

```
Process $i$'s program
**Shared**:
    $Version[1..\infty]$, $Try[1..\infty]$: arrays of booleans, initially all 0
**Local**:
    $v\_index$: integer, initially 1
    $leader$: process id, initially nil

1   repeat forever
2       Remainder
3       $Try[i] := 1$
4       repeat forever
5           while $Version[v\_index] = 1$ do $v\_index := v\_index + 1$ od
6           $leader := LeaderElect(v\_index)$
7           if $leader = i$ then break fi
8           await $((Version[v\_index] = 1) \vee (Try[i] = 0))$
9           if $Try[i] = 0$ then break fi
10      end repeat
11      Critical Section
12      $Try[i] := 0$
13      while $Version[v\_index] = 1$ do $v\_index := v\_index + 1$ od
14      if $Try[Enum(v\_index)] = 1$ then $Try[Enum(v\_index)] := 0$ fi
                                             /* help process $Enum(v\_index)$, */
15      else $Version[v\_index] := 1$ fi              /* or move to next version */
16  end repeat
```

Figure 9: (Non-adaptive) starvation-free mutual exclusion for unbounded concurrency using bits.

Theorem 5.2 *For solving starvation-free mutual exclusion, an infinite number of atomic bits and test&set bits are necessary and sufficient when the concurrency level is unknown bounded.*

Proof: In [Pet94], it is proved that n atomic registers and test&set bits are necessary for solving the starvation-free mutual exclusion problem for n processes (with concurrency level n). This implies the necessary condition above. The sufficient condition follows immediately from Theorem 4.7. ▮

5.1 Naming

The following theorem demonstrates that, in certain cases, a problem is solvable assuming unknown bounded concurrency, but is not solvable assuming unbounded concurrency. The *naming* problem, defined in Section 1, is to assign unique names to initially identical processes.

Theorem 5.3

1. *For unknown bounded concurrency, an infinite number of T&S bits are necessary and sufficient for solving wait-free naming.*
2. *For unbounded concurrency, there is no solution for wait-free naming, even when using an infinite number of T&S bits.*

Proof: Part 1. The sufficient condition follows from the following simple algorithm, a variation of algorithms due to Alur and Taubenfeld [AT96] and to Moir and Anderson [MA95]. The algorithm uses an infinite number of bits with initial values 0, indexed 1,2,.... Each process scans the bits, in order, starting with the first bit (i.e., bit number 1). At each step, the process applies the `test&set` operation, and either moves to the next bit if the returned value is 1, or stops when the returned value is 0. The process is assigned the name equal to the bit on which its (last) `test&set` operation returned 0. (Notice that wait-freedom is satisfied assuming unknown bounded concurrency, however, it is not satisfied assuming unbounded concurrency since there is a run in which some process will always return 1's and never terminates.)

To prove the necessary condition, we use the following simple argument. Consider an algorithm which solves the problem. Pick a process, run it alone until it acquires a name. Then pick another process and run it alone until it acquires a name, and so on. Since the processes must be assigned unique names, also each state of the algorithm at the time a process terminates must be different from any other such state. Thus, the state space must be infinite and hence also the number of bits.

Part 2. Assume to the contrary that such an algorithm, say A, exists. Let p and p' be two processes. We reach a contradiction by constructing an infinite run α of A in which p and p' always execute the same steps, and hence can never acquire different names. We construct α inductively such that, for each positive integer i, α has a prefix $\alpha(i)$ where:

1. p and p' have taken i steps each and their steps are exactly the same.
2. none of the test&set operations performed by p and p' have ever returned 0.

We define $\alpha(0)$ to be the empty run. Assume that we have constructed $\alpha(i)$. We show how to extend it to $\alpha(i+1)$. If the next operation of p and p' in $\alpha(i)$, is either a *reset* or a test&set on a bit which is already set to 1, then $\alpha(i+1)$ is constructed by letting p and p' each take one step, and we are done. Otherwise, their next operation is a test&set operation on a bit, say b, which is set to 0. We first extend $\alpha(i)$ by letting some other processes take steps until b is set to 1 and only then let p and p' take one step each, as before.

Setting b to 1 using other processes is the tricky part. We use the following simple observation. At any point in a computation, if $i+1$ identical processes apply a test&set operation to the same bit, then the values returned to the last i processes will be 1.

Consider the following extension of $\alpha(i)$. First we activate $i+1$ new identical processes q_1,...,q_{i+1}. Then, for $j = 1, ..., i, i+1$ schedule a round in which processes q_j,...,q_{i+1} each take a step, in order. At the end of the i-th round, process q_{i+1} has the same state as processes p and p'. This is because, like p and p', process q_{i+1} has taken exactly i steps and all of its test&set operations have returned 1. In the last round, process q_{i+1} sets b to 1. Finally, we complete the construction of $\alpha(i+1)$ by letting p and p' each take one step.

Note that, in $\alpha(i+1)$, there are only a finite number of active processes, although the number of active processes may be larger than in $\alpha(i)$. Thus the algorithm satisfies unbounded concurrency. ▮

6 Stronger Objects

6.1 RMW bits

A read-modify-write object supports a single operation, which atomically reads a value of a shared register, and based on the value read, computes some new value and assigns it back to the register. When assuming a fault-free model with required participation, many problems become solvable with small constant space. For example, as indicated in the second table in Section 1, it is trivial to implement consensus using a single, three-valued read-modify-write object. Mutual exclusion, however, is another matter:

Theorem 6.1

- *When only a finite number of RMW registers are used for solving starvation-free mutual exclusion with unknown bounded concurrency, one of them must be of unbounded size.*
- *One unbounded size RMW register is sufficient for solving first-in, first-out adaptive symmetric mutual exclusion when the concurrency level is unbounded.*

Proof: It is shown in [B+82] that in a model which supports a read-modify-write operation any starvation-free mutual exclusion algorithm for n processes requires at least $\sqrt{2n} + \frac{1}{2}$ values. This result implies the necessary condition.

To prove the sufficient condition, we slightly modify the ticket algorithm from [F+89]. The algorithm in Figure 10 works as follows: A process wishing to enter its critical section takes the next available ticket and waits until its ticket becomes valid, at which point it can safely enter its critical section. When it exits, it discards its ticket and validates the next invalid ticket in order (even if this ticket has not been taken yet).

```
Process i's program
Shared:
    (ticket, valid): integers, initially (0, 0)
Local:
    (ticket_i, valid_i): integers
1   $\langle (ticket_i, valid_i) := (ticket, valid)$
2     $ticket := ticket + 1\rangle$
3   while $ticket_i \neq valid_i$ do
4         $\langle valid_i := valid\rangle$ od
5   critical section
6   $\langle valid := valid + 1\rangle$
```

Figure 10: FIFO mutual exclusion using read-modify-write: The ticket algorithm

The shared register contains two fields, *ticket* and *valid*, each a non-negative integer. The brackets $\langle\ \rangle$ are used to explicitly mark the beginning and end of exclusive access to the shared read-modify-write register. An execution of a bracketed section is considered as an atomic action. Each process first reads the value of the shared register, stores its components in local memory, and increments *ticket* by one. At any later point, a process becomes the first in the "waiting line" if it learns (by inspecting *valid*) that its ticket number $ticket_i$ equals *valid*, in which case it can safely enter its critical section. ▮

6.2 Semaphores

Given the results so far about starvation-free mutual exclusion, it is natural to ask whether it can be solved with a bounded number of semaphores. The answer, as presented in [FP87], is that using weak semaphores, it can be solved with small constant space for unbounded concurrency.

A binary semaphore is a shared objects that may take the values 0 or 1, and is initially set to 1. It support two operations, called P and V:

- When a process performs a P operation, if the value is 1, then the value is set to 0; otherwise, the process is blocked until the value is 1. Testing and decrementing the semaphore are executed atomically.
- When a process performs a V operation, the value is set to 1.

The result below uses *weak* semaphores, in which a process, say process p, that executes a V operation will not be the one to complete the next P operation on that semaphore, if another process is currently blocked at that semaphore. Instead, one of the blocked processes, other than p, is allowed to complete the next P operation on that semaphore.

Theorem 6.2 (Friedberg and Peterson [FP87]) *There is an adaptive symmetric solution for starvation-free mutual exclusion using two atomic bits and two weak semaphores, when the concurrency is unbounded.*

7 Related Work

In [GMT2001], wait-free computation using only atomic registers is considered. It is shown that bounding concurrency reveals a strict hierarchy of computational models, of which unbounded concurrency is the weakest model. Nevertheless, it is demonstrated that adaptive versions of many interesting problems (collect, snapshot, renaming) are solvable even in the unbounded concurrency model. Wait-free randomized consensus algorithms for an unbounded number of processes using only atomic registers have been explored in [ASS2002], where it is also observed that standard universal constructions based on consensus continue to work with an unbounded number of processes with only slight modifications. In [MT2003], a general construction is presented, which implements wait-free consensus for an unbounded number of processes from any known solution for wait-free consensus (deterministic or randomized) for only finitely many processes. In [CM2002], the active disk paxos protocol is implemented for infinitely many processes. The implementation makes use of a solution to the consensus problem with an unbounded number of processes. The solution is based on a collection of a finite number of read-modify-write objects with faults that emulates a new reliable shared memory abstraction called a ranked register.

In [F^+89], it is shown that infinite space is required for solving starvation-free mutual exclusion even when using read-modify-write registers. However, Friedberg and Peterson [FP87] have shown that, using objects, such as semaphores, that allow waiting, enables a solution using two registers and two weak semaphores (see Theorem 6.2) [FP87]. In [BMT95], various types of shared counters are implemented in a model where the concurrency level is unknown bounded, and where noting can be assumed about the identities of the processes that might access the counters. Wait-free solvability of tasks when there is no upper bound on the number of participating processes has also been investigated [GK98] (see also [Gafni2002]). An underlying, automata-theoretic formulation of unbounded concurrency may be found in [L^+93, Lyn96].

All previously published mutual exclusion algorithms using atomic registers assume that the number of processes are bounded and a priori known. The question of whether there exists an adaptive mutual exclusion algorithm using atomic registers was first raised in [MT93], where an adaptive algorithm is presented for a given working system, in which process creation and deletions are rare. In [AT92], it is proven that no adaptive algorithm exists when time is measured by counting all accesses to shared registers. Hence we use another time complexity measure (defined previously), called system response time [PF77], for evaluating the complexity of adaptive mutual exclusion algorithms. Other complexity measures were also considered in the literature, for example counting only the number of remote memory references [AK2000], or assuming that busy-waiting is counted as a single step [AB2002, AST99].

In [CS94], the only previously known adaptive mutual exclusion algorithm was presented, in a model where it is assumed that the number of processes is bounded and a priori known. The algorithm exploits this assumption to work in bounded space, and does not work assuming unbounded concurrency. In [Lam87], an algorithm is presented which provides fast access in the absence of contention. However, in the presence of any contention, the winning process may have to check the status of all other n processes (i.e. access n different shared registers) before it is allowed to enter its critical section. As pointed out in [MA95], the algorithm from [Lam87] makes use of a shared object called a *splitter*. (This object is shown in Figure 4.) More recent results about adaptive mutual exclusion algorithms, for the known bound concurrency model, include [AB2002, AK2000, AST99, AST2002, CS94, Tau2004].

In [Pet94], upper bounds are presented for n-process mutual exclusion in a model which supports a read-modify-write operation. It is shown that a memoryless starvation-free solution can be achieved with $\lfloor n/2 \rfloor + 2$ values. Also several algorithms are presented using various types of semaphores (unfair, weak and strong), and it is proved that any starvation-free mutual exclusion algorithm for n processes using only atomic registers and test-and-set bits must use at least n atomic registers and test-and-set bits. This last result is used in proving the necessary condition of Theorem 5.2.

A symmetric election algorithm is presented in [SP89], for n processes (with concurrency level n), which uses only three atomic registers. The authors also showed that when participation is not required, only $\lceil \log n \rceil + 1$ registers are necessary and sufficient for solving the election problem. In addition, in [SP89], Styer and Peterson have proved that n registers are necessary and sufficient for deadlock-free symmetric mutual exclusion, while $n + 2\lceil \log n \rceil + 1$ registers are sufficient for starvation-free symmetric mutual exclusion. Finally, they proved that $2n - 1$ registers are necessary and sufficient for *memoryless* starvation-free symmetric mutual exclusion. Symmetric algorithm for a general function evaluation problem are presented in [AGM98].

An important space lower bound which shows that any deadlock-free mutual exclusion algorithm for n processes must use at least n shared registers, is proven in [BL93]. An $\Omega(\log_w n)$ lower bound on the contention-free time complexity of any algorithm that uses atomic registers, where w is the maximum number of processes that may simultaneously write the same register, is proven in [YA96]. Contention-free time complexity is defined in Section 2.3.1. In [AT96], it is shown that, even without contention, in any mutual exclusion algorithm for n processes, a process must access $\Omega(\sqrt{n})$ shared bits before entering its critical section. Many known mutual exclusion, consensus and other synchronization algorithms are discussed in detail in [Tau2006].

8 Discussion

We have explored how various assumptions about the number of processes, the concurrency level, and the number of participating processes, affect the design of shared memory algorithms. In particular, we have looked at the case of computing with an unbounded number of processes. There are many interesting questions that are left open: In the fault-free model using atomic registers only, are there interesting problems that can be solved for any bounded and *a priori* known number of processes but cannot be solved for a bounded and unknown number of processes (even with unbounded space)? Are there interesting problems that can be solved using a *finite* number of atomic registers by an unbounded number of processes with unknown bounded concurrency? Designing wait-free algorithms for problems such as collect, renaming and snapshot assuming unbounded concurrency are challenging problems. More recent work in this area demonstrates the existence of a hierarchy of computable tasks, determined by concurrency bounds [GMT2001].

Are there automatic transformations between some algorithms (such as symmetric algorithms) which work correctly for a finite number of processes, to algorithms which work correctly (perhaps assuming unknown bounded concurrency) for an unbounded number of processes?

The relationship between algorithms for an unbounded number of processes and adaptive algorithms is intriguing. Algorithms designed for an unbounded number of processes are often adaptive, but are there natural conditions under which one type of algorithm implies the other? For example, consider the following symmetric algorithm for unbounded concurrency: Take a fixed enumeration, $p_1, \ldots$ of process id's. Each process p_i writes the id's from the enumeration one at a time to a different shared register r_i, until process p_i's id is reached. This silly algorithm is not adaptive– the step complexity of even a solo execution is unbounded, as it depends on the position i of p_i's id in the enumeration.

Finally, are there algorithms that are correct assuming known bounded concurrency, but which fail if this constraint is not met? Can we, in general, modify such algorithms to guarantee at least safety (if not liveness) even when the constraints on the number of processes and the concurrency level are not met?

Acknowledgement We wish to thank an anonymous referee for many suggestions and corrections, and notably, catching an error in the conference version of the algorithm in Figure 3. We also thank the editor Faith Ellen and Nancy Lynch for many suggestions and corrections.

References

[Ang80] D. Angluin. Local and global properties in networks of processes. In *Proceedings of the 12th ACM symposium on Theory of Computing*, pages 82-93, 1980.

[AB2002] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. *Distributed Computing*, 15(3):177–189, 2002.

[ABNDPR90] H. Attiya and A. Bar-Noy and D. Dolev and D. Koller and D. Peleg and R. Reischuk. Renaming in an asynchronous environment. *Journal of the Association for Computing Machinery*, 37(3):524-548, 1990.

[AK2000] J.H. Anderson and Y. Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th international symposium on distributed computing*, 2000.

[AST99] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *Proc. 40th IEEE Symp. on Foundations of Computer Science*, pages 262–272, October 1999.

[AST2002] Y. Afek and G. Stupp and D. Touitou. Long-lived adaptive splitter with application. *Distributed Computing*, 15(2):67–86, 2002.

[AT92] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, December 1992.

[AT96] R. Alur and G. Taubenfeld. Contention-free complexity of shared memory algorithms. *Information and Computation* 126:1 (1996) 62–73. (Also in PODC 1994.)

[AGM98] H. Attiya, A. Gorbach, and S. Moran. Computing in totally anonymous asynchronous shared memory systems. In *Proc. 12th International Symposium on Distributed Computing*, LNCS 1499:49-61, September 1998. Also in: Information and Computation, 173(2):162–18, March 2002.

[ASS2002] J. Aspnes, G. Shah, and J. Shah, Wait-free consensus with infinite arrivals. In *Proc. 34th Annual Symp. on Theory of Computing*, 524–533, May 2002.

[ASW88] H. Attiya and M. Snir, and M. Warmuth. Computing on an anonymous ring. *Journal of the ACM*, 35(4):845–875, Oct. 1988.

[BMT95] H. Brit, S. Moran, and G. Taubenfeld. Public data structures: counters as a special case. *Proc. 3rd Israel Symposium on Theory of Computing and Systems,* Tel Aviv, 98–110, January 1995. Also in: Theoretical Computer Science, 289(1):401–423, 2002.

[B^+82] J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, and G. L. Peterson. Data requirements for implementation of N-process mutual exclusion using a single shared variable. *Journal of the Association for Computing Machinery*, 29(1):183–205, 1982.

[BL93] J. N. Burns and N. A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[CM2002] G. Chocker and D. Malkhi. Active disk paxos with infinitely many processes. In *Proc. 21th ACM Symp. on Principles of Distributed Computing*, 78–87, July 2002.

[CS94] M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[FL87] G. N. Frederickson and N. A. Lynch. Electing a leader in a synchronous ring. *Journal of the Association for Computing Machinery*, 34(1):98-115, 1987.

[F^+89] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. on Programming Languages and Systems*, 11(1):90–114, January 1989.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FP87] S. A. Friedberg and G. L. Peterson. An efficient solution to the mutual exclusion problem using weak semaphores. *Information Processing Letters*, 25(5):343–347, 1987.

[GK98] E. Gafni and E. Koutsoupias. On uniform protocols. `http://www.cs.ucla.edu/~eli/eli.html`, 1998.

[Gafni2002] E. Gafni. A Simple Algorithmic Characterization of Uniform Solvability. In *43rd Symp. on Foundations of Computer Science*, 228–237, 2002.

[GMT2001] E. Gafni, M. Merritt, and G. Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proc. 20th ACM Symp. on Principles of Distributed Computing*, 161–169, August 2001.

[Her91] M. P. Herlihy. Wait-free synchronization. *TOPLAS*, 13(1):124–149, 1991.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[L$^+$93] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic transactions.* 1993, Morgan Kaufmann.

[Lyn96] N. Lynch. *Distributed algorithms.* 1996, Morgan Kaufmann.

[MA95] M. Moir and J. Anderson. Wait-Free algorithms for fast, long-lived renaming, *Science of Computer Programming* 25(1):1–39, 1995.

[MT93] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993. (Also published as an AT&T technical memorandum, May 1991.)

[MT2000] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. *Proceedings of the 14th International Symposium on Distributed Computing*, LNCS 1914, 164–178, October 2000.

[MT2003] M. Merritt and G. Taubenfeld. Resilient Consensus for Infinitely Many Processes. *Proceedings of the 17th International Symposium on Distributed Computing*, LNCS 2648, 1–15, October 2003.

[Pet94] G. L. Peterson. New bounds on mutual exclusion problems. Technical Report TR68, University of Rochester, February 1980 (Corrected, Nov. 1994).

[PF77] G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proc. 9th ACM Symp. on Theory of Computing*, pages 91–97, 1977.

[SP89] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, 177–191, 1989.

[Tau2004] G. Taubenfeld. The black-white bakery algorithm. In *18th international symposium on distributed computing*, LNCS 3274, 56–70, October 2004.

[Tau2006] G. Taubenfeld. Synchronization Algorithms and Concurrent Programming. *Pearson / Prentice Hall*, ISBN 0-131-97259-6, 423 pages, 2006.

[YA96] J-H. Yang and J.H. Anderson. Time/Contention Trade-Offs for Multiprocessor Synchronization. *Information and Computation*, 124(1):68–84, 1996.