Computing with Infinitely Many Processes

under assumptions on concurrency and participation*

Michael Merritt†   Gadi Taubenfeld‡

August 28, 2002

Abstract

We explore four classic problems in concurrent computing (election, mutual exclusion, consensus, and naming) when the number of processes which may participate is infinite. Partial information about the number of actually participating processes and the concurrency level is shown to affect the possibility and complexity of solving these problems. We survey and generalize work carried out in models with finite bounds on the number of processes, and prove several new results. These include improved bounds for election when participation is required (even for finitely many processes, as investigated by Styer and Peterson [SP89]) and a new adaptive starvation-free mutual exclusion algorithm for unbounded concurrency. We 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 infinite process case.

*A preliminary version of the results presented in this paper appeared in *Proceedings of the 12th 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 Open University and the Interdisciplinary Center, Israel; and AT&T Labs. gadi@cs.openu.ac.il.

1 Introduction

1.1 Motivation

We explore several classic problems in concurrent computing (election, mutual exclusion, consensus, and naming) when the number of processes which may participate is (denumerably) infinite. Partial information about the number of actually participating processes and the concurrency level is shown to affect the possibility and complexity of solving these problems. This paper surveys and generalizes work carried out in models with finite bounds on the number of processes, and proves several new results. These include improved bounds for election when participation is required (even for finitely many processes, as investigated by Styer and Peterson [SP89]) and a new adaptive starvation-free mutual exclusion algorithm for unbounded concurrency. We 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 infinite process case.

Processes: In most work on the design of shared memory algorithms, it is assumed that the number of processes is finite and *a priori* known. Here we investigate the the design of algorithms assuming no *a priori* bound on the number of processes. In particular, we assume the number of active processes may be infinite. 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 finite, algorithms designed for an infinite number of processes may scale well: their time complexity may depend on the actual contention and not on the total number of processes.

A starvation-free mutual exclusion algorithm is known in this model, using two registers and two weak semaphores (see Theorem 5.2) [FP87]. Wait-free solvability of tasks when there is no upper bound on the number of participating processes has also been investigated [BMT95, GK98], but in this earlier work no run has an infinite number of participating processes. A model in which processes are continuously being created and destroyed (and hence their number is not *a priori* known) has also been considered [MT93]. (An underlying, automata-theoretic formulation of infinite concurrency may be found in e.g. [M$^+$93, Lyn96].)

Concurrency: 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. (That is, participating in the algorithm at the same instant of time. This is often called *point contention.* A weaker possible definition of concurrency, often called *interval contention*, is not considered here.) We distinguish between the following concurrency levels:

- *finite:* There is a finite bound (denoted by c) on the maximum number of processes that are simultaneously active, over all runs. (The algorithms in this paper assume that c is known.)
- *bounded:* In each run, there is a finite bound on the maximum number of processes that are simultaneously active.

- *unbounded:* In each run, the number of processes that are simultaneously active is finite but can grow without bound.
- *infinite:* The maximum number of processes that are simultaneously active may be infinite.

The case of infinite concurrency raises some issues as to the appropriate model of computation and semantics of shared objects. We assume that an execution consists of a (possibly infinite) sequence of *group steps*, where each group step is itself a (possibly infinite) sequence of steps, but containing at most *one* primitive step by each process. Hence, no primitive step has infinitely many preceding steps by any single process. The semantics of shared objects need to be extended (using limits) to take into account behavior after an infinite group step. This leaves some ambiguity when a natural limit does not exist. For example, what value will be returned by a read which follows a step in which infinitely many processes write different values? However, natural limits exist for all the executions we consider in this model. In particular, in the algorithm in Figure 7, the only write steps assign the value 1 to shared bits (which have initial value 0). The well-ordering of primitive steps in an execution of this model assures that there is a first write to any bit, and all subsequent reads will return 1.

Participation: When assuming a fault-free model with *required participation* many problems are solvable using only constant space. In this model, every process must eventually execute its code. It had been considered, for example, for the symmetric, finite process model 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.

We use the notation $[\ell, u]$-*participation* to mean that at least ℓ and at most u processes participate. Thus, for a total of n processes (where n might be infinite) $[1, n]$-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 $\ell \leq \ell' \leq u' \leq u$.

Thus, any solution assuming that participation is not required, is correct also for the case when participation is required, and hence it is expected that such solutions (for the case where participation is not required) may be less efficient and harder to construct.

1.2 Properties

We define two algorithm properties considered in the paper.

Adaptive algorithms: An algorithm is *adaptive* if the time complexity of processes' operations is bounded by a function of the actual concurrency. The term *contention sensitive* was first used to describe such algorithms [MT93], but later the term *adaptive* become commonly used. (Time complexity is computed using the standard model, in which each primitive operation on an objects is assumed to take no more than one time unit. In the case of mutual exclusion algorithms, we measure the maximum time between releasing the critical section, until the critical section is re-entered. For models in which there is a minimum bound on the number of participating processes, we measure one time unit to the first point at which the minimum number of processes have begun executing the algorithm.)

Symmetric algorithms: An algorithm is *symmetric* if the only way for distinguishing processes is by comparing (unique) process identifiers. Process id's can be written, read, and compared, but there is no way of looking inside any identifier. In particular, identifiers cannot be used to index shared registers. Various 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.)

1.3 Summary of results

We assume the reader is familiar with the definitions of the following problems:

1. The mutual exclusion problem, which is to design a protocol that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65];
2. The consensus problem, which is to design a protocol in which all correct processes reach a common decision based on their initial opinions [FLP85];
3. The (leader) election problem, which is to design a protocol where any fair run has a finite prefix in which all processes (correct or faulty) commit to some value in $\{0, 1\}$, and exactly one process (the leader) commits to 1. Although we require all processes to terminate in fair runs, we do not require the leader to be identified to the other processes. In a model with infinitely many 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 the id's of participating processes are not initially known. For asymmetric algorithms, we assume that process identities are natural numbers.
4. The *wait-free naming* problem which is to assign unique names to initially identical processes. Every participating process is able to get a unique name in a finite number of steps regardless of the behavior of other processes.

We show that even with a fixed 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; however, a result of Peterson ([Pet94]) implies that the more complex problem of starvation-free mutual exclusion (bounded concurrency, participation not required) still requires infinite space. In fact, even using read-modify-write registers to solve this problem, a result of Fischer *et al* ([F+89]) implies that infinite space is required. However, Friedberg and Peterson ([FP87]) have shown that using objects such as semaphores that allow waiting enables a solution with only constant space.

When there is no upper bound on the concurrency level, 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 (wait-free) naming assuming unbounded concurrency. However, naming can be solved assuming bounded concurrency using test&set bits only, hence it separates bounded from unbounded concurrency. At the boundary of unbounded and infinite concurrency, Friedberg and Peterson ([FP87]) have shown a separation for starvation-free mutual exclusion using registers and weak semaphores: two of each suffice for unbounded concurrency, but the problem cannot be solved using these primitives for infinite 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 finite and known. We use the following abbreviations: DF for deadlock-free; SF for starvation-free; mutex for mutual exclusion; U for Upper bound; L for Lower bound; RW for atomic read/write registers; T&S for test&set bits; wPV for weak semaphores; and RMW for read-modify-write registers. (The default is "No" for the adaptive and symmetric columns. All lower bounds hold for the most general, non-adaptive and asymmetric case.)

Problem			Model		Result					
	Name	Bound	Concurrency $c > 1$	Participation	Properties: adaptive?	Properties: symmetric?	Space: #	Space: size	Thm #	Using results from
1	Election	L_1	c	$[c, \infty]$			∞ state space		2.1	
		U	c	$[c, \infty]$			∞	2	2.2	
	(Upper	U	2	$[2, \infty]$	Y	Y	1	∞	2.3	
	bounds	U	c	$[c, \infty]$	Y	Y	2	∞	2.4	
	hold	U	c	$[c, \infty]$	Y		1	∞	2.5	
	also	U	c	$[1, \infty]$		Y	$\log c + 1$	∞	3.1	[SP89]
	consen-	L_2	c	$[1, \infty]$			$\log c + 1$	2	3.1	[SP89]
	for	U	c	$[\ell, \infty]$			$\log(c - \ell$	∞	3.4	
	sus.)			$\ell < c$			$+1)$ $+2$			
2	DF mutex	U	c	$[1, \infty]$		Y	c	∞	3.1	[SP89]
		L	c	$[1, \infty]$			c	2	3.1	[BL93]
3	Election SF mutex	L	bounded	$[1, \infty]$			∞	2	3.1	$1L_2$,2L
		U_1	unbounded	$[1, \infty]$	Y	Y	∞	∞	3.2	
		U_2	infinite	$[1, \infty]$			∞	2	3.3	
4	Implement Test & Set	L	bounded	$[1, \infty]$			∞	2	4.1	3L,5U
		U	unbounded	$[1, \infty]$	Y	Y	∞	∞	4.1	$3U_1$
		U	infinite	$[1, \infty]$			∞	2	4.1	$3U_2$

Results using atomic registers in a fault-free model

Problem			Model				Result					
	Name	Bound	Concurrency $c > 1$	Participation	Fault tolerance	Shared objects	Properties: adaptive?	Properties: symmetric?	Space: #	Space: size	Thm #	Using results from
5	Election DF mutex	L	c	$[\infty, \infty]$	0	T&S			1	2		trivial
		U	infinite	$[1, \infty]$	∞	T&S	Y	Y	1	2		trivial
6	SF mutex	L	bounded	$[1, \infty]$	0	RW,T&S			∞	2	4.2	[Pet94]
		U	infinite	$[1, \infty]$	0	RW,T&S			∞	2	3.3	$3U_2$
7	Naming	U	bounded	$[1, \infty]$	∞	T&S	Y	Y	∞	2	4.3	[AT96, MA95]
		L	bounded	$[1, \infty]$	∞	T&S	Y	Y	∞	2	4.3	
		L	unbounded	$[1, \infty]$	∞	T&S			no alg		4.3	
8	Consensus	L	2	$[\infty, \infty]$	1	RMW			2	2		trivial
		U	infinite	$[1, \infty]$	∞	RMW	Y	Y	1	3		trivial
9	SF mutex	L	bounded	$[1, \infty]$	0	RMW			1	∞	5.1	[F$^+$89]
		U	unbounded	$[1, \infty]$	0	RMW	Y	Y	1	∞	5.1	[B$^+$82]
		U	infinite	$[1, \infty]$	0	RMW			∞	2		$3U_2$
10	SF mutex	U	unbounded	$[1, \infty]$	0	RW,wPV	Y	Y	2,2	2	5.2	[FP87]
		L	infinite	$[1, \infty]$	0	RW,wPV			no alg		5.2	[FP87]
Results using stronger shared objects												

2 Atomic registers: participation is required

This section demonstrates that requiring a minimum number of process to participate (the required participation model) 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 state space. We begin by showing that any solution to election in this model requires infinite state 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.

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

Even when participation is required, (and equal to the concurrency) any solution to election for infinitely many 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 2.1 *There is no solution to election with finite 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 (possibly infinite) solo run $solo(p)$ of each process p, and define $write(p)$ to be the subsequence of write events in $solo(p)$. 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, $repeat(p)$ will necessarily 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 steps, as follows: Ignoring the states of the processes for now, let $\beta(p) = x_0, x_1, x_2 ...$, where the x_j are the successive states of the shared memory in $\beta(p)$. Suppose x_j is the first state that repeats in $\beta(p)$, and that x_k is the last state in $\beta(p)$ such that $x_j = x_k$. 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 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. The solo runs of any subset of $C = p_1, ..., p_c$ of these processes are compatible: by scheduling appropriately, we can construct runs in which each of the processes take the same steps as in their solo runs. By an easy case analysis, in one such run no leader is elected, or there are many runs in which two or more leaders are elected, a contradiction.

The (compatible) run of the processes in C is constructed as follows: Let $\gamma = y_1, ..., y_k$. First run these c process until each one of them is about to execute its first write. Run process p_1 until it takes the last step in $\beta(p_1)$ that leaves the memory in state y_1. Repeat for p_2 through p_c, then repeat this loop for each state in γ. The resulting finite run leaves the shared memory in state s, and each of the c processes p_i has taken 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.

Notice that the signatures are used only in the first stage of the construction, when the processes run until the memory state is s, in such a way that each process "thinks" it ran solo to get there. After reaching (together) the recurring state s, each process can be run solo until s recurs. Since the same state s recurs infinitely many times for each of the processes in C, we can schedule each in turn that has not terminated to take (some finite, nonzero number) of steps until s occurs next, then schedule the next one (i.e., round robin scheduling). Now everyone either runs forever or terminates as in a solo run. ∎

2.2 Algorithms for election and consensus for concurrency c

The previous theorem shows that election for infinitely many processes requires unbounded shared memory. When participation is required, many problems become solvable. 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 2.2 *For any finite 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

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

if Marked = 0 then
    (Leader, Marked) := (i, 0)
    await (Leader ≠ i) or (Marked = 1)
    local_leader := Leader
    (Leader, Marked) := (local_leader, 1)
fi
return(Leader)
```

Figure 1: Symmetric election for concurrency 2

not to miss any bit that is set, it scans the array according to a diagonalization: a schedule that visits each bit an infinite number of times. (A canonical example is to first read $b[1]$; then $b[1], b[2]$; then $b[1], b[2], b[3]$, etc.) Once a process notices that c bits are set, it set $b[0]$ to 1. The process with the smallest id among the c that are set to 1 is the leader. By scanning the bits after reading $b[0]$ as 1, the other processes can also learn the id of the leader. This solution for election can be trivially modified, using one other bit, to solve consensus. ∎

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 2.3 *For finite 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. ∎

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 2.4 *For any finite 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 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 values, comparing it to the set they have already seen, and writing when they know a value not reflected in the set read. This phase ends when a process reads c different values. (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 below, the second register is used both to "gate" access to the *Union* register, and to break symmetry on the set of process id's. It contains two fields: a process id (in the *Leader* field) and a bit (the *Marked* field). Processes access the *Union* field only if they read the *Marked* field as 0. 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 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 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 on the first c processes imposed by set inclusion on their *local_union2* variables. This order evolves as these variables are updated–because these sets are only updated by union, processes can only move upward in this order. (And call the set of id's in *local_union2* the set that the associated process *knows about*.) As long as *Marked* is not set, we argue that the only stable state of this partial order 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 state. It contains minimal elements: processes with $|local\_union2| < c$ and such that no other process knows about strictly fewer processes. Once a minimal process writes its value of *local_union2* to *Union*, no strictly smaller set can be written. Hence, each of these processes can write at most once without the partial order changing, as any read must then be of the same value they wrote. But if the partial order isn't changed

```
Process i's program.
Shared:
        (Leader, Marked): (Process id, boolean), initially (0, 0)
        Union: set of at most c process id's, initially ∅
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}

if Marked = 0 then (Leader, Marked) := (i, 0) fi
local_union1 := Union
while (|local_union1| < c) ∧ (Marked = 0) do
    if ¬(local_union2 ⊆ local_union1) then
        Union := local_union1 ∪ local_union2
    fi
    local_union2 := local_union1 ∪ local_union2
    local_union1 := Union
od
local_leader := Leader
(Leader, Marked) := (local_leader, 1)
return(Leader)
```

Figure 2: Symmetric election for concurrency c

otherwise, eventually a process that knows a different id will write to the *Union* register and then be read by a minimal process.

It follows that at some point a process will learn that c processes are participating (i.e., $|Union| = c$) and set *Marked* to 1, allowing the rest of the processing to terminate. 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 the set of active processes 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 process are not initially known to all processes, the trivial solution where all processes choose process 1 as a leader is excluded.

Theorem 2.5 *For any finite 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 2.4). As before, each of the first c processes repeatedly updates the shared *Union* register with any

```
Process i's program
Shared:  (Union, Sig, Ack): Union a set of at most c process id's,
             0 ≤ Sig ≤ c, Ack ∈ {0,1}, initially (∅, 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

(local_union1, local_sig1, local_ack1) := (Union, Sig, Ack)
while |local_union1| < c do
    if ¬(local_union2 ⊆ local_union1) then
        (Union, Sig, Ack) := (local_union1 ∪ local_union2, 0, 0)
    fi
    local_union2 := local_union1 ∪ local_union2
    (local_union1, local_sig1, local_ack1) := (Union, Sig, Ack)
od
local_union2 := local_union1
leader := min{q : q ∈ local_union1}
if i ∉ local_union1 then
    return(leader)
elseif i ≠ leader then
    myrank := |{h ∈ local_union1 : h < i}|
    while local_sig1 < c do
        if (|local_union1| < c) ∧ (acked = 0) then /* Register is stale, haven't acked. */
            (Union, Sig, Ack) := (local_union2, 0, 0)
        elseif (local_sig1 = myrank) ∧ (local_ack1 = 1) then
            (Union, Sig, Ack) :=(local_union2, myrank, 0) /* Acknowledge signal. */
            acked := 1
        fi
        (local_union1, local_sig1, local_ack1) :=(Union, Sig, Ack)
    od
    return(leader)
else                                        /* |local_union2| = c, i = leader */
    while local_sig1 < c do
        if (|local_union1| < c) or (local_sig1 < local_sig2) then
            (Union, Sig, Ack) := (local_union2, local_sig2, local_ack2)/* Register is stale. */
        elseif local_ack1 = 0 then                  /* Signal acknowledged */
            local_sig2 := local_sig2 + 1
            (Union, Sig, Ack) := (local_union2, local_sig2, 1) /* Send new signal. */
        fi
        (local_union1, local_sig1, local_ack1) := (Union, Sig, Ack)
    od
    return(i)
fi
```

Figure 3: (Asymmetric) election for concurrency c

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. The key difficulty is termination detection among the first c processes, using the single register. This is accomplished by establishing an explicit handshake between the leader and each of the other $c-1$ initial processes.

In outline, once a process p observes the size of the set in *Union* to be at least c (we argue below exactly c), it knows the identity of the leader (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. Having acknowledged the signal, p stops participating in the updates to *Union*. (At this point, process p knows that the leader knows all c initial id's, and will eventually refresh *Union* if necessary.) The leader itself initiates each handshake 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.

As in the previous algorithm, no process exits the while loop in line 2 until c processes have entered the loop. (In a given run, let *Initial* denote the set of these initial c processes.) We argue below that none of the initial processes terminates until the shared register contains the final and stable value $(Initial, c, 1)$. 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 it knows the id's of c participating processes, (the initial while loop in line 2) and indeed, until it also knows that the leader knows the id's of c participating processes (the update in line 16 that is skipped once the leader's signal is acknowledged). As in the argument in the preceding algorithm, this implies that all processes in *Initial* will eventually learn the id's in *Initial*.

By induction on the rank $r \geq 1$ of process p in *Initial*, 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 $(Initial, c, 1)$ and terminate. The other of the 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. ∎

3 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 are more difficult to construct. The main new result of this section is a new adaptive starvation-free mutual exclusion algorithm for unbounded concurrency.

3.1 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 a finite number of processes establishes some bounds:

Theorem 3.1 *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 infinite 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 infinite 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. ∎

Theorem 3.1 implies that when the concurrency level is not finite, an infinite number of registers are necessary for solving the election and mutual exclusion problems. But are an infinite number of registers sufficient? (We observe, based on a result from [YA94], that if we restrict the number of processes that can try to write the same register at the same time, the answer is no.)

3.2 Mutual exclusion algorithms for unbounded concurrency

Theorem 3.1 leads naturally to the question of whether an infinite number of registers suffice for solving mutual exclusion for unbounded concurrency, when participation is not required. 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 3.2 *There is an adaptive symmetric solution to starvation-free mutual exclusion for unbounded concurrency (and hence also 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 below. This algorithm has the following interesting properties:

1. It works for unbounded concurrency.

2. In the absence of contention, only nine accesses to the shared memory are needed (eight in the entry code and one in the exit code).

3. It is adaptive – its time complexity (even in the worst case, measured from 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 the next subsection, 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.

3.2.1 Splitters

In [Lam87], Lamport presented a fast 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. Each process that invokes the splitter moves either *down*, *right* or *wins*. The properties of Lamport's splitter are listed below.

Proposition 3.1 *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. 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 so. 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 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. ∎

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

```
Process i's program

x: integer (the initial value is immaterial)
y: boolean, initially 0

1 x := i
2 if y = 1 then move right fi
3 y := 1
4 if x ≠ i then move down
5 else win fi
```

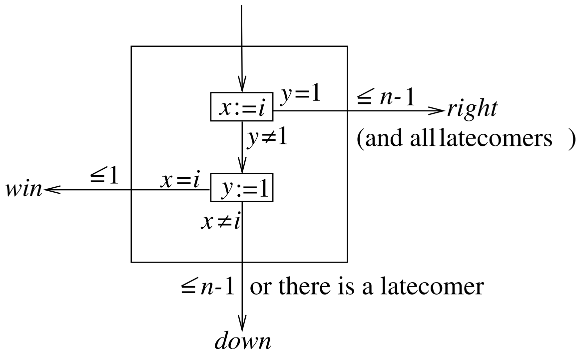

Figure 4: *Lamport's splitter.*

Proposition 3.2 *The new splitter in Figure 5 satisfies the first four properties of Lamport's splitter (as listed in Proposition 3.1), 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. The second property of Proposition 3.1 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 the 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 new splitter is a *latecomer* in the embedded Lamport's splitter. The third and fourth safety properties follow immediately from the corresponding conditions in Proposition 3.1. 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 Proposition 3.1, exactly one process (*early* in both splitters) *wins* in Lamport's splitter, and then in the new splitter. (Since b is never set to 1.) ∎

```
Process i's program

x: integer (the initial value is immaterial)
b, y, z: boolean, initially 0

x := i
if y = 1  then  b := 1 move right fi
y := 1
if x ≠ i  then  await (b = 1) or (z = 1)
                if z = 1 then move right
                else move down fi
else   z := 1
       if b = 0 then win
       else move down fi fi
```

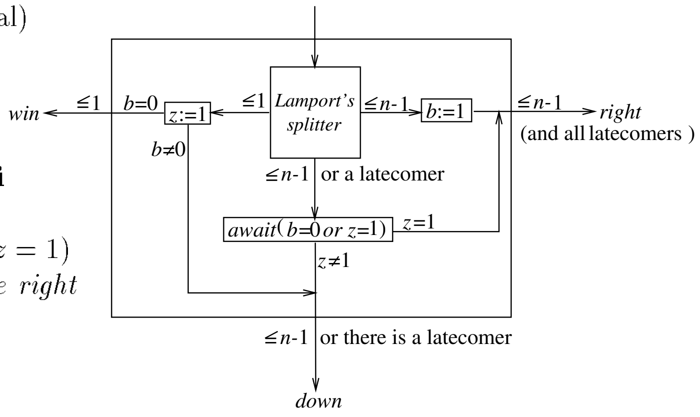

Figure 5: The code of the *new splitter*.

3.2.2 An adaptive mutual exclusion algorithm

Using the properties of 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:

Proposition 3.3 *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 re-join) the competition starting from that level.

By the properties of the new splitter, 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 greater index, and wait for the global pointer *next* 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 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 eight accesses to shared atomic registers are needed before entering the critical section. ∎

```
Process i's program
Shared:
     next: integer, initially 0
     x[0..∞]: array of integers (the initial values are immaterial)
     b[0..∞], y[0..∞], z[0..∞]: array of boolean, initially all 0
Local:
     level: integer, initially 0
     win: boolean, initially 0

     start: level := next
     repeat
        x[level] := i
        if y[level] then b[level] := 1        /* move right */
                 await level < next
                 goto start fi
        y[level] := 1
        if x[level] ≠ i then await (b[level] = 1) or (z[level] = 1)
                 if z[level] = 1 then await level < next /* move right */
                   goto start
                 else level := level + 1 fi     /* move down */
        else z[level] := 1
          if b[level] = 0 then win := 1              /* win */
          else level := level + 1 fi fi         /* move down */
     until win = 1
     critical section
     next := level + 1
```

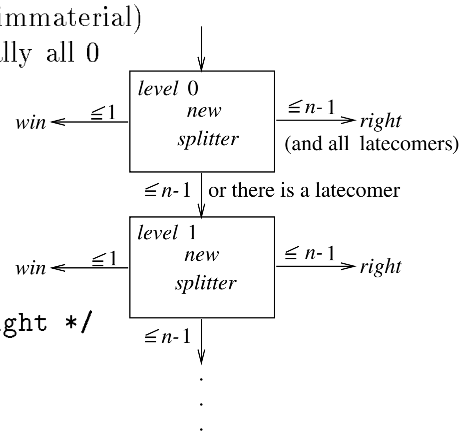

Figure 6: Adaptive deadlock-free mutual exclusion for unbounded concurrency.

3.2.3 Proof of Theorem 3.2

Proof: The adaptive deadlock-free algorithm of Figure 6 is easily modified using standard "helping" techniques to satisfy starvation-freedom. That is, processes set flags when they leave the remainder section, and before leaving the critical section, a process examines the flag of the next process (in some enumeration) and grants the critical section if it determines the associated process is waiting. Because the number of processes is infinite, a global diagonalization is necessary instead of a round-robin schedule: a process helps others in the order given by an enumeration in which every process id appears infinitely often. (Global variables recording progress in this globalization are maintained via the mutual exclusion of the critical section.) Starvation-freedom follows.

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. ∎

The question of designing an adaptive mutual exclusion algorithm was first raised in [MT93]. The solution presented there is efficient provided process creation and deletions are rare (the term *contention sensitive* was suggested but the term *adaptive* become commonly used).

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

    RaceOwner[i] := 1
    if RaceOther[i] = 0 then Win[i] := 1 else Lose[i] := 1 fi
    repeat forever
        RaceOther[index] := 1
        if RaceOwner[Index] = 1 then
            await (Win[Index] = 1 or Lose[Index] = 1)
        fi
        if Win[index] = 1 then return(index) fi
        index := index + 1
    end repeat
```

Figure 7: (Non-adaptive) leader election for infinite concurrency using bits

In [CS94], the only previously known adaptive mutual exclusion algorithm was presented, in a model where it is assumed that the number of processes (and hence concurrency) is finite. The algorithm exploits this assumption to work in bounded space, and does not work assuming unbounded concurrency. In [Lam87], a fast 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. Few recent results about adaptive mutual exclusion algorithms for finite and a priori known number of processes appear in [AB2002, AK2000, AST2002].

3.2.4 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, it suffices to use only atomic bits.

Theorem 3.3 *There is a non-adaptive asymmetric solution to election, consensus and starvation-free mutual exclusion for infinite concurrency using an infinite number of bits.*

Proof: Figure 7 presents a simple algorithm for election. Modification to solve consensus is trivial. 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 minimum winner is the leader. The other two bits are used by the owner 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. The first process i to perform the read in the second line 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.

Agreement: In an infinite run, there is a minimum i such that $Win[i]$ is ever set. 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. So every process will return i.

A fairly straightforward revision implements deadlock-free mutual exclusion. That is, make a copy of these bits for every entrance to the critical section. Each copy has a control bit. The leader is the process who enters the critical section. The exit code is to set the control bit. 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 index to the current copy of the algorithm to determine who to help next in a (predetermined) diagonalization. ∎

3.3 Relating participation and concurrency

Section 2 focused on the case in which the number of required participants, ℓ, is equal to the concurrency, c. Note that if $c < \ell$, the notion of required participation depends on termination of at least some of the first c participants. Indeed, the leader election algorithms of the previous section will work correctly for this case, as no process terminates until c have begun, and the leader is determined. If $\ell < c$, the situation is more complex. The following theorem unifies the results of Theorems 2.5 (participation is required) and 3.1 (participation is not required), demonstrating a relation between participation and concurrency in this case.

Theorem 3.4 *For concurrency c with $[\ell, \infty]$-participation where $l < c$, $\log(c - \ell + 1) + 2$ registers are sufficient for solving election.*

Proof: The algorithm is a combination of the a single register assuming $[c, \infty]$-participation (from Theorem 2.5), and of the one using $\log c + 1$ registers assuming $[1, \infty]$-participation (from Theorem 3.1). First, we use the single register algorithm (Theorem 2.5) as a filter. Here, instead of choosing a single leader, up to $c - \ell + 1$ processes may be elected. These processes continue to the next level. To implement this, we slightly modify the single register algorithm. A process needs to wait only until it notices that ℓ (instead of c) processes are participating, and if it is the biggest among them it continues to the next level. Thus, at most $c - \ell + 1$ (and at least one) processes continue to the next level. In that level, they compete using the algorithm of Theorem 3.1 (no change is needed in that algorithm), until one of them is elected. This level requires $\log(c - \ell + 1) + 1$ registers. ∎

4 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 4.1 *An infinite number of atomic registers are necessary for implementing a test&set bit when concurrency is bounded, and sufficient when concurrency is infinite.*

Proof: To prove the necessary condition assume to the contrary that such an implementation is possible with a fixed, 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 to deadlock-free mutual exclusion with bounded concurrency using a finite number of atomic registers. This contradicts Theorem 3.1. The sufficient condition follows immediately from Theorem 3.3. ∎

Theorem 4.2 *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 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 3.3. ∎

4.1 Naming

The following theorem demonstrates that in certain cases a problem is solvable assuming bounded concurrency, but is not solvable assuming unbounded concurrency. The *wait-free naming* problem is to assign unique names to initially identical processes. After acquiring a name the process may later release it. A process terminates only after releasing the acquired name. 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 (such as abnormal termination).

Theorem 4.3

1. *For 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 to 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 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 terminate.)

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. To prove the necessary condition, 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 recursively 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 of p and p' has 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. If $i+1$ identical processes apply the same operation to the same bit, then the returned values for at least i processes must be the same. Now consider the following extension of $\alpha(i)$. First we activate $i+1$ new processes and let each one of them take one step. Since the processes are initially identical, each will apply the same operation to the same bit and hence at least i of them will get the same response. Thus, i processes are still identical. Next we activate these i identical processes and let each one take one step, and are guaranteed to end up with a set of at least $i-1$ identical processes. We repeat this procedure i times, until one process, say q, has taken i steps. We claim that q is identical to both p and p'. This follows from the fact that, as for p and p', process q has taken exactly i steps, and none of the test&set operations of q has ever returned 0.

We now activate q and let it set b to 1, and finally complete the construction of $\alpha(i+1)$ by letting p and p' each take one step. ∎

5 Stronger Objects

5.1 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:

Process i's program

Shared:

$(ticket, valid)$: integers, initially $(0, 0)$

Local:

$(ticket_i, valid_i)$: integers

$\langle (ticket_i, valid_i) := (ticket, valid)$
$\quad ticket := ticket + 1 \rangle$
while $ticket_i \neq valid_i$ **do**
$\qquad \langle valid_i := valid \rangle$ **od**
critical section
$\langle valid := valid + 1 \rangle$

Figure 8: FIFO mutual exclusion using read-modify-write: The ticket algorithm

Theorem 5.1

- *When only a finite number of RMW registers are used for solving starvation-free mutual exclusion with 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 operations any starvation-free mutex 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 8 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).

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. ∎

5.2 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, but not with infinite concurrency level.

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 refers to *weak* semaphores, in which a process that executes a V operation will not be the one to complete the next P operation on that semaphore, if another process has been blocked at that semaphore. Instead, one of the blocked processes is allowed to "pass" the semaphore to a blocked process.

Theorem 5.2 (Friedberg and Peterson [FP87])

- *There is an adaptive symmetric solution to starvation-free mutual exclusion using two atomic bits and two weak semaphores, when the concurrency is unbounded.*
- *There is no solution to the starvation-free mutual exclusion problems using any finite number of atomic registers and weak semaphores, when the concurrency is infinite.*

6 Discussion

We have explored how various assumptions about the number of processes, the concurrency level, and the number of participating processes, effect the design of shared memory algorithms. In particular, we have looked at the case of computing with infinitely many 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 finite and *a priori* known number of processes but cannot be solved for a finite 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 infinite number of processes with bounded concurrency? Designing wait-free algorithms for problems such as collect, renaming and snapshot assuming unbounded concurrency are challenging problems. 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 bounded concurrency) for infinitely many processes?

Finally, are there algorithms that are correct assuming finite 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 The authors 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 Nancy Lynch for catching some additional errors.

References

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

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

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

[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.

[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: accepted to Theoretical Computer Science.)

[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.

[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.

[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.

[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.

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

[LA87] M. C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

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

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

[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.)

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

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

[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.

[YA94] J-H. Yang and J.H. Anderson. Time bounds for mutual exclusion and related problems. In *Proc. 26th ACM Symp. on Theory of Computing*, 224–233,May 1994.