Naming Symmetric Processes Using Shared Variables

Ömer Eğecioğlu and *Ambuj K. Singh* *

Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106

Abstract

Implementations of inter-process communication and synchronization in distributed systems usually rely on the existence of unique ids for the processes. We consider the problem of generating such ids for identical processes in a shared-variable system. A randomized protocol that assigns distinct ids to the processes within an expected polynomial number of rounds using a polynomial number of boolean atomic variables is presented.

Keywords: Parallel algorithm, randomized protocol, synchronization.

1 Introduction

Breaking the symmetry of a set of processes is a fundamental problem in distributed systems. It is the cornerstone of inter-process communication and synchronization as techniques for achieving these usually rely upon the fact that the processes are not symmetric [15, 21]. Symmetry may be broken in a number of ways: by assigning distinct ids to the processes, by assigning different programs to the processes, by instituting an asymmetric communication network, or by assuming distinct initial states for the processes [11]. In this paper, we assume indistinguishable programs, identical initial states for the processes, and consider the task of introducing asymmetry through the generation of distinct process ids.

The complexity of any solution to the problem of breaking symmetry depends on the parameters of the underlying system model: synchronous versus asynchronous, deterministic versus randomized, or fine-grain versus coarse-grain atomicity. Symmetry can be broken easily if we assume coarse-grain atomicity provided by primitives such as *test-and-set* or *read-and-increment*. This is because such primitives can be used to serialize access to a variable, and consequently unique ids can be generated based on the order in which the processes access the variable. For example, if a process can read and modify a shared-variable in a single step, then symmetry can be broken trivially through a single shared-variable: every process reads and increments this shared-variable and uses the value read as its id. Upon termination, every process has a distinct id irrespective of the initial contents of the shared-variable. In this paper we focus on the case of fine-grain atomicity, i.e., we assume that in a single atomic step a process can only read or write at most one shared-variable.

If the model is synchronous and deterministic (in the sense that randomization is not allowed), then it is fairly straightforward to see that symmetry cannot be broken: a set of indistinguishable processes

*Work supported in part by NSF grants CCR-9008628 and CCR-9223094.

executing identical programs can only read and write the same values in the shared and local memory in each time step, since all processes necessarily execute identical operations. Therefore, the processes remain indistinguishable from one another. This observation implies that the problem does not admit a deterministic solution in an asynchronous model either. This is because of the possibility of *lock-step* executions (i.e., executions in which a process takes its $(k+1)$-st step only after all the remaining processes have completed their k-th step). The net outcome of such lock-step executions is identical to the synchronous case and therefore for the same reasons outlined earlier, symmetry cannot be broken.

It follows from the above discussion that randomization must be used to solve the problem in either a synchronous or an asynchronous model. To introduce randomization, we assume a primitive $Random(1 \cdots m)$ that returns a random integer in the range 1 to m. In the synchronous model, the problem of assigning distinct ids from the set $\{1, 2, \ldots, N\}$ to N given indistinguishable processes can be solved by the algorithm in Figure 1. In addition to a few private variables, the processes use a shared variable X, which is an array of $m \geq N$ bits. In an iteration of the **repeat** loop of procedure *Toss*, each process resets its old choice $X[l]$, chooses a new random index value l and sets $X[l] := 1$. Then it counts the number of ones in the array X and exits the loop upon finding this count equal to K.

```
Program Main:
    var shared X : array [1 .. m] of 0|1;
    var private id, i, l, count : integer;
                success : boolean;

    initially id = ⊥;            /* id is undefined */
              l = 1;

    Procedure Toss(K):
        begin
            1:    for i := 1 to m do  X[i] := 0 od;
            2:    repeat
            3:        X[l] := 0;
            4:        l := Random(1 ··· m);
            5:        X[l] := 1;
            6:        count := 0;
            7:        for i := 1 to m do  count := count + X[i] od;
            8:        success := (count = K)
            9:    until success;
            10:   id := l            /* assign id */
        end Toss;

    begin      /* Main */
        Toss(N)
    end
```

Figure 1: Randomized Symmetry Breaking in a Synchronous Model

Note that the randomized algorithm in Figure 1 terminates when each of the N processes has

chosen and set a distinct bit of the array X. Consequently the processes will have distinct ids upon termination. Since the probability of all processes making distinct random choices is non-zero during each iteration of the **repeat** loop, processes will eventually find *success* to be true and terminate. The expected termination time of the algorithm can be shown to be exponential for $m = N$ and $O(N^2)$ for $m = cN^2$.

In light of the above observations, the main body of our paper investigates the only remaining case of the problem of symmetry breaking: that of randomized solutions in an asynchronous model using fine-grain atomicity.

The organization of the paper is as follows. In Section 2, we present the relevant background and previous work on symmetry breaking and related problems in distributed systems. In Section 3, we present definitions and a formal statement of the problem. Section 4 contains the description of the main algorithm which is constructed in a general setting. As we develop various procedures used in the algorithm, we record the the requirements they need to satisfy. The implementation of symmetry breaking under varying assumptions on the atomicity of shared variables (composite registers, unbounded atomic variables, and bounded atomic variables) is then given in Section 5. In each of these implementations the correctness follows by verifying that the requirements set forth in Section 4 are met. In Section 6 we analyze the time and space complexity of our algorithms. Finally, in Section 7 some concluding remarks are presented. Detailed proofs of propositions that are used in the body of the paper can be found in the appendix.

2 Previous Work

A substantial amount of research has been devoted to anonymous message-passing networks where the identity of the processes is not known and inter-process communication is by message passing. Angluin addressed the question of how much each process must know in order to carry out useful functions such as setting up a spanning tree or electing a leader [5]. She showed that for a certain class of graphs such as a ring of four processes, symmetry cannot be broken deterministically. This work was later extended by Yamashita and Kameda [31]. They consider the leader election problem (among other problems) and characterize networks based on the attributes that need to be known to each node in order to elect a leader. The kinds of network attributes that they consider include an upper bound on the number of processes, the exact number of processes, and the exact network topology.

Given the above impossibility results, Itai and Rodeh considered randomization techniques for solving problems over anonymous networks [18]. They showed that in the case of a ring topology, a leader election algorithm that always terminates can be constructed if and only if the number of processes in the ring is known within a factor of two. Subsequently, a number of authors including Scheiber and Snir [29] and Matias and Afek [26] have presented randomized algorithms for leader election on arbitrary networks.

Symmetry breaking and related problems have also been studied on shared-variable systems. Lehmann and Rabin [23] have shown that the Dining Philosophers Problem [16] cannot be solved by deterministic protocols for processes that are symmetric. Burns examined symmetry in the context of the Mutual Exclusion Problem [15] and defined the notions of *strong* and *weak* symmetry [10]. He also showed that there do not exist any strongly symmetric solutions to the problem. Johnson and Schneider [19] proposed the idea of *similarity* in distributed systems and investigated solutions to problems such as leader election for different kinds of atomic primitives. A number of other authors have considered the Consensus Problem [3, 13, 17] and shown consensus to be deterministically im-

possible if only atomic variables are used. Abrahamson presented a randomized protocol for achieving consensus with atomic variables [1]. The expected number of steps of this algorithm is exponential in the number of processes. Subsequently, Aspnes and Herlihy improved upon this result and presented an algorithm with a polynomial expected running time [6] if the counters used are allowed to be unbounded. Recently, Attiya, Dolev, and Shavit improved Aspnes and Herlihy's algorithm so that the counters remain bounded in size [8]. Another problem that has been studied in this area is the Choice Coordination Problem [28]. This problem consists of a set of processes and a set of alternatives, and the aim is for the processes to agree on one of the alternatives. Rabin [28] and Bar-Noy, Ben-Or, and Dolev [9] have presented deterministic as well as randomized solutions to the problem using test-and-set instructions.

Our work is motivated in large part by a paper by Lipton and Park [25] in which the shared-variable model that we assume here is laid out, as well as papers by Teng [30] and Lim and Park [24]. Lipton and Park present an elegant randomized algorithm that ensures that if the algorithm terminates then the processes have distinct ids. The authors also consider optimal organization of shared variables in order to increase the probability of termination. Optimal organization of shared variables is further investigated by Teng, and by Lim and Park. We extend the previous work by presenting algorithms that ensure both safety (i.e. if the algorithm terminates, then the processes have distinct ids), and progress (i.e. the algorithm terminates within an expected finite number of rounds) conditions. Our final algorithm uses shared bits. Recently, Kutten, Ostrovsky, and Patt-Shamir [20] have independently proposed an efficient solution using larger shared variables.

3 Preliminaries and Problem Statement

In this section we first outline a simple model for discussing and reasoning about concurrent programs, and later present a formal statement of the symmetry breaking problem.

Our formal system model is that of a *fair transition system* [12, 27]. A *concurrent program* consists of a set of processes, a set of variables, an initial state, and a fairness rule. A *process* is specified by the means of a program text and consists of a set of actions. Each variable in a program is classified as either *private* or *shared*. A variable is private if it can be accessed by at most one process; otherwise it is shared. The program counter of a process is a private variable. We do not assume any special initial values for the shared variables. Each action of a process obeys fine-grain atomicity, i.e., each action either reads a shared variable, writes a shared variable, obtains a random value, or performs some local computation. We assume programs executed by the processes to be indistinguishable, in particular there is no reference to a process's identity in the text.

A *state* is an assignment of values to the variables (both shared and private) of the program. It defines which actions of the program can possibly be executed next. We refer to such executable actions as being *enabled* at that state. A process is said to be enabled at a state provided one of its actions is enabled at the state. For simplicity, we assume that at most one action per process is enabled at a state. If the execution of an action A in a state s leads to a state t, then we express this by $s \stackrel{A}{\rightarrow} t$. We use an interleaving semantics and denote a *history* as a sequence $s_0 \stackrel{A_0}{\rightarrow} s_1 \stackrel{A_1}{\rightarrow} \cdots$, where s_0 is the initial state.

More than one process may be enabled at a state of a given history. The selection of the process that takes the next step is determined *a priori* and is oblivious to the actual execution or the values of the shared variables. This is essentially the assumption of an *oblivious adversary*. This assumption is necessary as symmetry cannot be broken if the adversary is allowed to be *adaptive* [20]. On account of this, repeated random choices in a history can be assumed to be independent of each other. In

order to ensure progress, we also assume that a history is *fair*, i.e., continuously enabled actions are eventually executed.

We reason about the correctness of our programs with the help of two operators: *unless* and $\mapsto$ (pronounced *leads-to*) [12]. For any two predicates p and q, we say that the property p *unless* q holds in a program if and only if for all histories, if predicate $p \wedge \neg q$ holds at any state then the next state satisfies $p \vee q$. Similarly, we say that the property $p \mapsto q$ holds in a program if and only if for all histories, if predicate p holds at any state then predicate q holds at the same or at a subsequent state.

Based on the *unless* operator, it is possible to define two additional program properties: *stable* and *invariant*. We say that the property *stable* p holds in a program if and only if p *unless false* holds, i.e., once predicate p holds at a state in a history, it continues to hold at all subsequent states. We say that the property *invariant* p holds in a program if and only if predicate p holds at the initial state and *stable* p holds. In other words, predicate p holds at all the reachable states of the program.

The time complexity of a synchronous system is usually measured by counting the number of steps taken by a single process. This measure does not apply to an asynchronous system as processes may be executing at different speeds. However, if we define a round to be a minimal interval over which every process which has not yet terminated executes at least one step, then counting the number of rounds provides one way of measuring the time complexity of asynchronous systems. This measure is called *round complexity* of an asynchronous computation. The reader is referred to [14] for further details.

The formal specification of the symmetry breaking problem for the case of N processes is as follows. Every process has a special private variable called id that is undefined initially (a special symbol $\perp$ denotes an undefined value) and is assigned at most once by the process. A solution is *correct* if and only if the following two requirements are met in all histories:

- *Safety:* For all processes p,

  a. $id_p \in \{1, 2, \ldots, N, \perp\}$,

  b. $id_p \neq \perp \;\Rightarrow\;$ process p is not enabled, and

  c. for any $q \neq p$, $(id_p \neq \perp \;\wedge\; id_q \neq \perp) \;\Rightarrow\; id_p \neq id_q$,

- *Progress:* For all processes p, $id_p \neq \perp$ within an expected finite number of rounds.

The first part of the safety requirement defines the range of values that variable id can assume. The second part states that the process terminates once it has computed an id. The last part of the safety requirement states that once assigned, the ids of processes are distinct. The progress requirement states that every process obtains an id within an expected finite number of rounds. The following two theorems establish necessary conditions for the existence of a solution to the problem.

Theorem 1 *Symmetry cannot be broken if the exact number of participating processes is not known.*

Proof Assume that a given solution that does not mention N works correctly when $N = k$ and also when $N = l$ with $k < l$. Consider the case when the number of participating processes equals k. Let P be the set of participating processes and let $p \in P$. Since the expected number of rounds for the assignment of id to each process is finite, there exists a history h which assigns distinct ids to the processes in P within a finite number of rounds. Consider processes $q_i, 1 \leq i \leq l - k$, that are not in P. Construct a history h' of processes in $P \cup \{q_i \mid 1 \leq i \leq l - k\}$ obtained from history h as follows:

whenever process p executes an action in h, we insert steps by processes $q_i, 1 \leq i \leq l-k$, in which they execute the same action as process p. Thus, processes p and $q_i, 1 \leq i \leq l-k$, execute the same sequence of actions, reading and writing the same shared and private variables with the same values. (In particular, these processes also choose the same random numbers.) History h' as constructed above is a legal history. Therefore, history h', now of l processes, also assigns unique ids to all the processes. However, since processes p and $q_i, 1 \leq i \leq l-k$, have executed identical actions, they also have the same id upon termination. This violates the safety condition. □

As a consequence of the above impossibility result, we assume that the number N of participating processes is known to all the processes. The next theorem develops another necessary condition for breaking symmetry.

Theorem 2 *The computation of unique ids requires cooperation among all processes.*

Proof We show that in any history h, all the processes necessarily write a shared variable that is read by some process before the computation of the first unique id. The proof of this claim is by contradiction. Suppose that there exists a history h in which process p computes a unique id after some prefix h_1 of h and assume that process q does not write any shared variable that is read by some process in h_1. Construct a prefix h_1' from h_1 as follows. First, delete all the actions of process q from h_1. Then, whenever process p executes an action in h_1, we insert a step by process q in which it executes the same action as process p. History h' is obtained by extending the prefix h_1' constructed above as needed by inserting steps of other processes. Since process q does not write any shared variable that is read by a process in h_1, process p performs the same sequence of actions in h_1 and h_1'. By construction, processes p and q execute the same sequence of actions in h_1'. Therefore, they obtain the same id in prefix h_1' and in history h'. This violates the safety condition. □

As a consequence of the above theorem, symmetry cannot be broken by *wait-free* processes [4, 17].

4 The Algorithm

In order to motivate an asynchronous solution to the problem, we first rewrite the synchronous algorithm of Figure 1 in the form shown in Figure 2. Subroutine *Initialize* in procedure *Toss* is implemented by setting all bits of array X to zeros as in line 1 of Figure 1; subroutine *Choose* is implemented by resetting the old choice and making a fresh choice on X as in lines 3-5 of Figure 1; subroutine *Scan* is implemented by counting the number of bits set as in lines 6-8 of Figure 1.

Let us define a scan operation to be successful if it finds *success* to be true, and to be useful if it is not interfered with by a choose or an initialize operation. The proof of progress in the synchronous case is based on the following two properties. First, every scan operation is useful on account of the synchrony, and second, the probability that a useful scan operation is successful is non-zero. However since *Choose*, *Scan*, and *Initialize* operations are not atomic, usefulness of scan operations cannot be guaranteed if the processes execute asynchronously. In order to achieve progress in an asynchronous setting, instead of requiring that every scan operation is useful, we require only that the number of useful scan operations increase in any history. The synchronous algorithm of Figure 2 is extended as follows: $Toss(K)$ now uses K copies of array X encoded as rows of an array A_K. Instead of a single choose operation on array X, a total of K choose operations are performed, one on each row of the array. Similarly, instead of one, K scan operations are performed in each iteration of the **repeat** loop. By executing the choose operations on the rows of A_K in the order $A_K[1], A_K[2], \ldots, A_K[K]$ and subsequently executing the scan operations in the reverse order $A_K[K], A_K[K-1], \ldots, A_K[1]$, we can show that every iteration of the **repeat** loop generates a useful scan.

```
Program Main:
    var shared X;
    var private i, l, count : integer;
                success : boolean;

    initially id = ⊥ ∧ l = 1;
    Procedure Toss(K):
        begin
            Initialize(X);
            repeat
                Choose(X, l);
                Scan(X, success)
            until success
        end Toss;

    begin      /* Main */
        Toss(N);
        id := l
    end Main
```

Figure 2: Rewritten Synchronous Algorithm

Since processes may now exit asynchronously from $Toss(K)$ upon executing a successful scan, we assume a procedure *Detect* that estimates the number of processes executing $Toss(K)$. A process keeps on executing $Toss(K)$ until it performs a successful scan or the number of processes is *detected* to be less than K. In the latter case, the processes repeat the same procedure on a different array of shared variables. The resulting algorithm is shown in Figure 3.

In place of the private variable l of the synchronous case, each process now has a private array $rand$ to record its last choices on the rows of array A_K. We assume that procedure $Initialize(K, i)$ initializes the ith row of array A_K. Procedure $Choose(K, i, rand[i])$ first resets the previous choice made by a process on row $A_K[i]$. Then it makes a fresh choice on the row and records the fresh choice in $rand[i]$. Procedure $Scan(K, i, success)$ scans row $A_K[i]$ for K distinct choices by the processes and sets *success* to true if and only if K distinct choices are observed. Procedure $Detect(count)$ returns in *count* an upper bound on the total number of processes that are executing *Toss* with any argument.

As in the synchronous case, processes begin execution of Figure 3 by invoking procedure *Toss* with argument $current := N$. Some process then eventually executes a successful scan and exits. The identity of the row on which the process succeeded is recorded in private variable *row*. On account of asynchrony, the remaining processes may not be able to execute a successful scan at the same time step. These processes eventually detect that $count < N$, and invoke *Toss* with argument $N - 1$. The same sequence of steps with parameter $current := N - 1$ is repeated on array A_{N-1} to break the symmetry among the remaining processes. It is also possible for $m > 1$ processes to be successful on array A_N. In that case, the remaining processes cascade down arrays $A_{N-1}, A_{N-2}, \ldots$, until they arrive at the correctly indexed array A_{N-m}. Now we have $N - m$ processes executing $Toss(N - m)$ on array A_{N-m}. This again leads to a successful scan. The process of convergence to the correctly indexed A array and completion of a successful scan continues until all the processes have executed a successful scan. A high level description of the possible state transitions of a process p executing *Main* is given in Figure 4. (For any private variable x, we use x_p to denote the variable specific to

```
Program Main:
    var shared A_K : array [1 ·· K] of rowtype,  1 ≤ K ≤ N;
    var private id, i, row, count, current : integer;
                success : boolean;
                rand : array [1 ·· N] of integer;

    initially count = N  ∧  id = ⊥  ∧  (∀i :: rand[i] = 1);

    Procedure Toss(K):
        begin
            for  i := 1 to K do  Initialize(K, i) od;
            repeat
                for  i := 1 to K do  Choose(K, i, rand[i]) od;
                row := K + 1;
                repeat
                    row := row − 1
                    Scan(K, row, success);
                until row = 0  ∨  success;
                Detect(count)
            until success  ∨  count < K
        end Toss;

    begin   /* Main */
        current := N + 1 ;
        repeat
                current := current − 1;
                Toss(current);
        until success
        ···;
        ···;
    end Main
```

Figure 3: Randomized Symmetry Breaking in an Asynchronous Model

process p.) A process exiting upon executing a successful scan waits until all the processes exit and then the processes cooperate to assign unique ids from the set $\{1, 2, \ldots, N\}$.

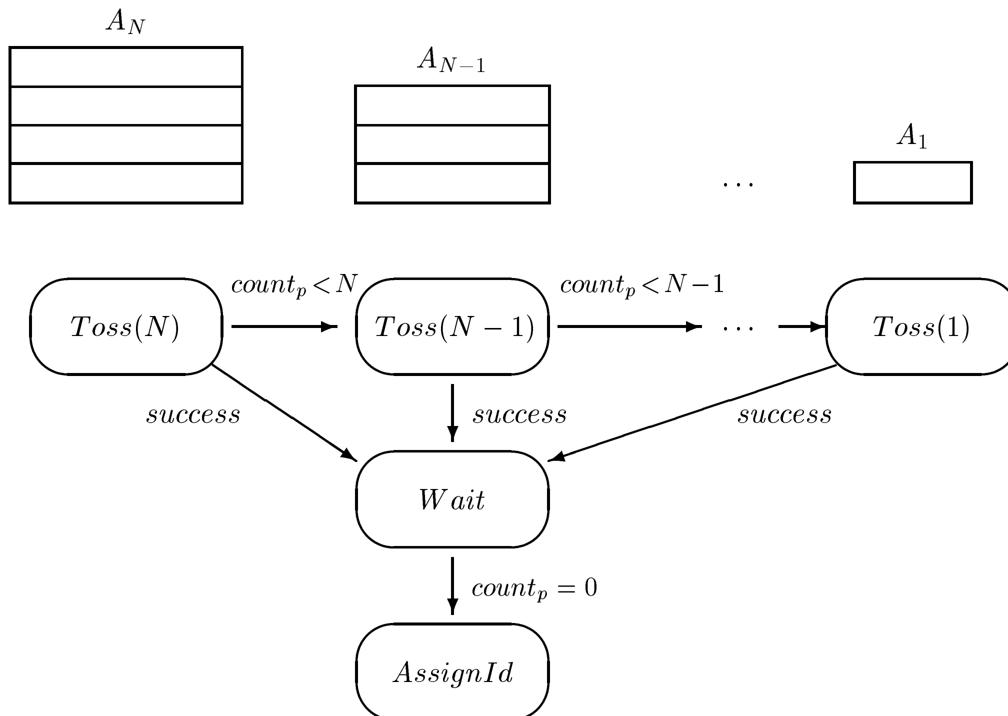

Figure 4: State Transitions for a process p.

In order for the general algorithm in Figure 3 to be correct, an obvious property that the four procedures *Initialize*, *Choose*, *Scan*, and *Detect* should satisfy is that they should terminate. Next, we consider additional requirements on these four procedures. Implementations satisfying the required properties under varying degrees of atomicity will be detailed later.

When K processes are executing $Toss(K)$ on array A_K, one of them should eventually execute a successful scan operation. We discuss this requirement in Section 4.1. In Section 4.2 we consider the issue of ensuring that all unsuccessful processes converge to the appropriately indexed A array, and repeat the symmetry breaking procedure there. This is followed by the method of distinguishing among successful processes, which is presented in Section 4.3. Section 4.4 outlines an implementation of procedure *Detect*, and finally Section 4.5 addresses the remaining problem of generation of distinct ids for the processes.

4.1 Ensuring Success for One Process

Here, we discuss the requirement that when K processes are executing $Toss(K)$ on array A_K, some process eventually executes a successful scan. First we introduce some terminology.

4.1.1 Notation and Definitions

Let $active(K)$ denote the set of processes that have started their execution of $Toss(K)$ and let $done(K)$ denote the set of processes that have completed the execution of $Toss(K)$. For any state

s, let *successful*(K) denote the set of processes in $done(K)$ that have performed a successful scan on array A_K prior to state s. Define

$$SUCCESSFUL = \bigcup_{K=1}^{N} successful(K) \ .$$

A process p is called *successful* if $p \in SUCCESSFUL$. Note that by definition

$$successful(K) \subseteq done(K) \subseteq active(K) \ ,$$

and that membership in all three sets is stable.

Let $choosing(p, K, i)$ hold at a state s if and only if process p is executing a choose operation on row $A_K[i]$ at state s. For any scan operation S on row $A_K[i]$, let *choosing-set*(S) denote the set of processes q for which the predicate $choosing(q, K, i)$ holds at some state during the execution of S. Let *comp-init*$_p(K)$ hold if K processes are in $active(K)$ and process p has completed at least one choose operation on each row of array A_K. Let the predicate *comp-init*(K) hold if $active(K)$ is not empty and *comp-init*$_p(K)$ holds for all processes in $active(K)$. In other words, *comp-init*(K) holds if and only if $active(K)$ contains K processes and all these processes have executed a choose operation on each row of array A_K. We will show later that $active(K)$ can contain at most K processes. Consequently, predicates *comp-init*$_p(K)$ and *comp-init*(K) are stable.

A scan operation S on a row of array A_K is formally defined to be *useful* if *comp-init*(K) holds immediately prior to S, and if *choosing-set*(S) is empty. The first conjunct in this definition implies that all K processes in $active(K)$ have completed their initializations and chosen a value on each row of array A_K. The second conjunct implies that no choose operations by any process in $active(K)$ interleave with the execution of the scan. We define the predicate $useful\text{-}scan(S)$ to be true if S is useful. As stated earlier, a scan operation S on a row of array A_K is defined to be *successful* if it sets *success* to true. We denote this by saying that the predicate *succ-scan*(S) holds.

4.1.2 Specification of *Detect*

In order to ensure that $active(K)$ contains at most K processes, procedure *Detect* should estimate the quantity $N - |SUCCESSFUL|$ by providing an upper bound on the number of processes that may be executing $Toss(K)$ for any K. This is stated as requirement $(DET1)$:

Requirement $(DET1)$:

$N - \lvert SUCCESSFUL \rvert \leq count_p$, for all p.

Condition $(DET1)$ is purely a safety requirement which guarantees that *Detect* never sets the variable *count* of a process to a value smaller than $N - |SUCCESSFUL|$. Note that since $|SUCCESSFUL|$ is non-decreasing, the quantity $N - |SUCCESSFUL|$ is non-increasing. We will add a progress component $(DET2)$ to the specification of *Detect* later. Based on property $(DET1)$, it is possible to prove the following two propositions (see Appendix):

Proposition 1 $|active(K)| \leq K$.

Proposition 2 $(|active(K)| = K \ \wedge \ |done(K)| > 0) \ \Rightarrow \ (\exists S :: succ\text{-}scan(S))$ *, where S ranges over scan operations on rows of A_K.*

The first proposition says that no more than K processes can be active on array A_K. The second proposition states that if K processes execute $Toss(K)$ and if one of these processes completes its execution of $Toss(K)$, then some process must have performed a successful scan S on some row of A_K.

4.1.3 Specification of $Initialize, Choose$, and $Scan$

In general, a *successful* scan operation need not be *useful* as it may observe the right number of distinct choices in spite of (or even because of) interferences by other initialize and choose operations. Similarly, a *useful* scan operation need not be *successful* as it may not observe the right number of distinct choices. However, we require that a useful scan operation during which all K processes in $active(K)$ have chosen and set different values in the appropriate row is necessarily successful. In other words, if S is a scan operation on row $A_K[i]$ then

Requirement ($SUC1$):

$useful\text{-}scan(S) \;\wedge\; [(\forall p,q : p,q \in active(K) : rand_p[i] \neq rand_q[i])$ immediately prior to $S] \;\Rightarrow$
$succ\text{-}scan(S)$.

4.1.4 Proof of Progress

Now we show that if procedures $Initialize, Choose, Scan$, and $Detect$ satisfy the requirements ($DET1$) and ($SUC1$) detailed above, then beginning from any state in which K processes are actively executing procedure $Toss(K)$, some process will be successful within an expected finite number of rounds. The proof of this can be motivated by noting that each history of the synchronous algorithm (Figure 2) either contains a successful scan or produces a sequence of useful scans $S_1, S_2, \ldots$ on array X. Since each of these useful scans observes a fresh set of random choices by the processes, success is guaranteed within an expected finite number of rounds. Proposition 3 asserts a similar fact about the asynchronous algorithm, the difference being that the sequence of useful scans constructed may now be over different rows of array A_K.

Proposition 3 *Starting from any state in which comp-init*(K) *holds, either a successful scan occurs on some row of* A_K*, or there exists an infinite sequence* $S_m, m \geq 0$*, of useful scans by processes in* $active(K)$ *such that*

1. *scan* S_m *is made on a row* r_m *of array* A_K*, and*
2. *each scan in the sequence observes the result of fresh random choices by the processes, i.e., for any two scans* S_{m_1} *and* S_{m_2} *with* $m_1 < m_2$*, if* $r_{m_1} = r_{m_2}$ *then each process in active*(K) *makes a fresh random choice on row* r_{m_1} *between scans* S_{m_1} *and* S_{m_2}*.*

Using propositions 2 and 3, we prove

Theorem 3 *Starting from any state in which* $|active(K)| = K$*, a state in which* $|successful(K)| > 0$ *is reached within an expected finite number of rounds.*

Proof Since procedures $Initialize, Choose, Scan$, and $Detect$ terminate,

$|active(K)| = K \;\mapsto\; comp\text{-}init(K) \;\vee\; (|active(K)| = K \;\wedge\; |done(K)| > 0)$.

From Proposition 2,

$(|active(K)| = K \;\wedge\; |done(K)| > 0) \;\Rightarrow\; (\exists S :: succ\text{-}scan(S))$.

Therefore,

$|active(K)| = K \ \mapsto\ comp\text{-}init(K)\ \vee\ (\exists S :: succ\text{-}scan(S))$.

Thus, eventually either $comp\text{-}init(K)$ holds or there is a successful scan on a row of A_K. In the latter case, some process will be successful and eventually belong to $successful(K)$. In the former case, Proposition 3 guarantees the existence of an infinite sequence of useful scans $S_m, m \geq 0$ such that scan S_m is on row r_m of array A_K, and each scan in the sequence observes the result of fresh random choices by the processes. Since a fresh random choice by a process is independent of those made previously, it follows just as in the synchronous case that within an expected finite number of rounds some useful scan operation S_m will observe distinct choices by the processes. By property $(SUC1)$, such a useful scan will be successful. Consequently, after an expected finite number of rounds some process will belong to $successful(K)$. □

4.2 Ensuring Success for All Processes

Theorem 3 states that once K processes start executing the procedure $Toss(K)$ in Figure 3, some process will be successful within an expected finite number of rounds and exit *Toss(K)*. When the number of processes executing $Toss(K)$ becomes less than K, the remaining processes should be able to detect this fact and exit $Toss(K)$ by finding $count < K$. To ensure this, we require that *Detect* set *count* to $N - |SUCCESSFUL|$. Accordingly, we add the following progress component to the specification of procedure *Detect*.

Requirement $(DET2)$:

Starting from a state in which $N - |SUCCESSFUL| = m$, repeated invocation of *Detect* by a process p will eventually set $count_p \leq m$.

The above property, in conjunction with Theorem 3 allows us to conclude that every process eventually executes a successful scan. First, we state the following proposition which is proved in the appendix.

Proposition 4 *Starting from any state in which* $p \in active(K)\ \wedge\ |SUCCESSFUL| = L$, *within an expected finite number of rounds*

$$p \in SUCCESSFUL\ \vee\ (p \in active(N - L - 1)\ \wedge\ |SUCCESSFUL| > L\ ).$$

Using Proposition 4 we obtain

Theorem 4 *Every process is successful within an expected finite number of rounds.*

Proof Let p be any process. From procedure *Main*, there exists a state s_0 at which process $p \in active(N)$. Let $|SUCCESSFUL| = m_0$ at state s_0. By Proposition 4, within an expected finite number of rounds, a state s_1 is reached at which either p has been successful and the desired condition is met, or $p \in active(N - m_0 - 1)\ \wedge\ |SUCCESSFUL| = m_1$, where $m_1 > m_0$. The above steps can be applied repeatedly to get a sequence of states $s_0, s_1, s_2 \ldots$ and a sequence of increasing values $m_0, m_1, m_2, \ldots$. Since each $m_i \leq N$, the theorem follows. □

4.3 Distinguishing between Successful Processes

So far we have shown that every process is successful within an expected finite number of rounds. In this section, we show how symmetry can be broken among successful processes. Consider any two

processes p and q such that $p \in successful(K)$ and $q \in successful(L)$. There are two cases to consider: $K \neq L$ and $K = L$.

In the first case, the symmetry between p and q can be broken by noting that a process belongs to at most one *successful* set. Distinguishing between processes p and q is more difficult when $K = L$, i.e., when both processes perform successful scans on rows of the same array A_K. One way to break symmetry in this case would be to require that all processes in $successful(K)$ perform successful scans on the same row as in the synchronous algorithm. However, it is not *necessary* for the processes to be successful on the same row. Since processes successful on different rows of array A_K can be distinguished based on the identity of the row, it suffices to ensure that all processes successful on the same row of A_K make distinct choices on that row. In order to achieve this, we require the procedures *Initialize*, *Choose*, and *Scan* to satisfy property ($SUC2$) stated below. (Symbol "$\setminus$" denotes set difference.)

Requirement ($SUC2$):

For any successful scan S on the $i-$th row of A_K, ($\forall p, q : p, q \in active(K) \setminus choosing\text{-}set(S) : rand_p[i] \neq rand_q[i]$) immediately prior to S.

Based on ($SUC2$), we can prove

Theorem 5 *All processes successful on the same row of array A_K make distinct final choices on that row.*

Proof Let $p_m, 1 \leq m \leq M$, be the processes successful on some row r of array A_K. Let $S_{p_m}, 1 \leq m \leq M$, be the corresponding successful scans by the processes on row r.

Consider the final choose operations executed by processes $p_m, 1 \leq m \leq M$, and let process p_l be the last process to complete its choose operation. Consider the scan S_{p_l}. According to the algorithm in Figure 3, $choosing(p_m, K, r)$ is false for each process p_m during the execution of S_{p_l}. Therefore by requirement ($SUC2$), $rand_{p_m}[r] \neq rand_{p_n}[r]$ for each pair (p_m, p_n) of processes. Consequently, all the processes make distinct choices on row r of array A_K. □

4.4 Implementing *Detect*

Besides ensuring termination, an implementation of procedure *Detect* should satisfy the properties ($DET1$) and ($DET2$). These requirements imply that an implementation of *Detect* should be able to estimate the size of the set $SUCCESSFUL$. Since $SUCCESSFUL = \bigcup_{K=1}^{N} successful(K)$ and the sets $successful(K)$ are disjoint, it suffices to estimate the size of each $successful(K)$. Theorem 5 provides a convenient way to estimate the size of the this set, since all processes that succeed on the same row of array A_K make distinct choices. If each process p that executes a successful scan on some row r of array A_K copies its most recent choice on that row to the corresponding row of another 2-dimensional array B_K, then array B_K will have exactly $|successful(K)|$ ones. The implementation of *Detect* can then count the number of ones in each row of B_K and use the total number of ones as an estimate of $|successful(K)|$, $1 \leq K \leq N$.

Procedure *Detect* is given in Figure 5. Variables $B_K, 1 \leq K \leq N$, are shared 2-dimensional arrays of bits. Each array B_K has K rows and the size of each row is given by an integer $rowsize(K) \geq K$. Variables $index, i$, and *ones* are private to *Detect*. The function $NumOnes(X, m)$ returns the number of occurrences of 1 in the input array X of size m. Its implementation is straightforward in terms of a loop that examines each entry in the array and is skipped for brevity. Procedure *Detect* uses this function to compute the total number of ones in the B arrays.

```
Procedure Detect(count):
    var private index, i, ones : integer;
        begin
            ones := 0;
            for index := 1 to N do
                for i := 1 to index do
                    ones := ones + NumOnes(B_index[i], rowsize(i))
                od
            od;
            count := N − ones
        end Detect
```

Figure 5: Procedure *Detect*

Procedure *Main* is now updated to ensure that each process that is successful on row r of array A_K records its final random choice on that row to array B_K. Note that the procedure *Toss* records the identity of the row on which a process executed a successful scan in variable *row*. Since we do not assume any initial values for shared variables, processes must first initialize each array B_K to all zeros. This initialization is done at the start of *Main* before any invocations of *Toss*. Initialization of the B arrays by one process may, however, erase bits set by another successful process. Therefore, a successful process needs to keep setting its choices until all processes have executed a successful scan. The modified procedure *Main* is shown in Figure 6. The descriptions of *Detect* and *Toss* are skipped for brevity.

Next, we prove that *Detect* satisfies requirements ($DET1$) and ($DET2$). If process p is successful on row r of array A_K then it sets the bit $rand[r]$ to one in row $B_K[r]$. Therefore,

$$ones \le \sum_{K=1}^{index} |successful(K)|$$

is an invariant of the outer **for** loop in procedure *Detect*. Since $\sum_{K=1}^{N} |successful(K)| = |SUCCESSFUL|$, and *count* is set to $N - ones$, we have $count \ge N - |SUCCESSFUL|$. Therefore procedure *Detect* of Figure 5 satisfies requirement ($DET1$).

For a proof that *Detect* satisfies requirement ($DET2$), consider any state s at which

$$N - \sum_{K=1}^{N} x_K = m \ ,$$

where $x_K = |successful(K)|$, $1 \le K \le N$. Let $x_{K,r}$ be the number of processes that succeeded on row r of array A_K. Thus,

$$x_K = \sum_{r=1}^{K} x_{K,r}.$$

The proof that *Detect* satisfies requirement ($DET2$) will follow if we can show that repeated invocation of *NumOnes* on a row r of array B_K by a process p eventually leads to a state t such that any further invocation of function *NumOnes* by process p after state t on row r of array B_K returns a value $\ge x_{K,r}$. To this end, consider repeated invocations of *NumOnes* by process p after state s. Let q be a

```
Program Main:
    var shared  A_K : array [1 ·· K] of rowtype,  1 ≤ K ≤ N;
                B_K : array [1 ·· K, 1 ·· rowsize(K)] of 0|1,  1 ≤ K ≤ N;
    var private current, id, i, j, l, row, count : integer;
                success : boolean;
                rand : array [1 ·· N] of integer;

    initially ...;

    Procedure Toss(K):
        ···;
        ···;
    Procedure Detect(count):
        ···;
        ···;

    begin   /* Main */
        /* initialize each array B_K */
        for l := 1 to N do
            for i := 1 to l do
                for j := 1 to rowsize(l) do B_l[i, j] := 0 od
            od
        od;
        /* end initialize */
        current := N + 1;
        repeat
            current := current − 1;
            Toss(current)
        until success;
        /* Process successful on A_current[row].
           Keep setting bit B_current[row, rand[row]] until count = 0 */
        repeat
            B_current[row, rand[row]] := 1;
            Detect(count)
        until count = 0

        AssignId(current, row, rand[row], id)

    end Main
```

Figure 6: Updated Procedure *Main*

process belonging to *successful*(K) that succeeded on row r. By construction of $Main$, process q will copy its final choices into row r of array B_K. Since requirement ($DET1$) is satisfied, $count_q$ remains greater than zero as long as process p keeps on activating *Detect* in order to exit on some A array. Therefore, on account of the final **repeat** loop in Figure 6, process q keeps on copying its choice to row r of array B_K as long as process p keeps on invoking *Detect* in order to exit on some A array. The bit set by process q can be erased only a finite number of times on account of initializations of array B_K due to other processes. Consequently, there exists a future state t_q after which the bit set by process q will not be erased. Let t be the last of these states t_q as q ranges over the processes in *successful*(K) that succeeded on row r. The bits set by these processes persist after state t and are furthermore distinct on account of Theorem 5. Consequently, any invocation of $NumOnes$ after state t by process p will return a value $\geq x_{K,r}$. This completes the proof that *Detect* satisfies requirement ($DET2$).

4.5 Computation of Unique Ids

Finally, we are ready to discuss the mechanism by which processes compute their ids. After a process p belonging to *successful*(K) finds $count_p = 0$ during the execution of the final **repeat** loop in Figure 6, it invokes procedure $AssignId$. The first two arguments to procedure $AssignId$ identify the index K and the row r of the array on which process p executed a successful scan. The third argument to procedure $AssignId$ contains the final random choice *val* made by p on that row. Procedure $AssignId$ shown in Figure 7 computes *id* using a lexicographic ordering as a sum of three values: sum_1, sum_2, and sum_3. Value sum_1 is the cumulative sizes of all successful sets with indices larger than K, i.e.,

$$sum_1 = \sum_{index=K+1}^{N} |successful(index)| \ ,$$

and serves to distinguish between processes that belong to different *successful* sets. Value sum_2 is the number of processes that exited on array A_K by executing a successful scan on a row preceding row r and distinguishes between processes that exit by performing successfuls scans on different rows of array A_K. Finally, sum_3 equals the number of processes that exited on array A_K by executing a successful scan on row r and chose a random value smaller than process p for the row. It distinguishes between processes that exit on the same row of array A_K.

The correct computation of sum_1, sum_2, sum_3 using the B arrays requires caution as some processes may be initializing the entries of the arrays to zero. When process p finds $count_p$ to be zero in procedure $Main$ prior to invoking $AssignId$, it has observed a bit set by every process and consequently, all processes have completed their initializations of the B arrays. This implies that no bit of the B arrays is reset to zero during the execution of $AssignId$. In order to ensure that all the appropriate bits of B arrays have been set to one, the invocation of $AssignId$ by process p first assigns $B_K[r, val] := 1$ and then waits until an invocation of *Detect* observes *count* to be zero. The entries of the B arrays can be shown to be stable henceforth and the computation of the ids can begin.

5 Implementations of *Initialize*, *Choose*, and *Scan*

In this section we focus on implementations of *Initialize*, *Choose*, and *Scan* under three levels of atomicity of shared objects: composite registers, unbounded atomic registers and bounded atomic registers. Composite registers [2, 4] are array-like variables in which every component can be written individually and the whole array can be read in a single step. The implementation of $Main$ and *Detect*

Procedure $AssignId(K, r, val, id)$:

    var private $index, i, j, sum_1, sum_2, sum_3$: **integer**;

        begin

          $B_K[r, val] := 1$;

          repeat $Detect(count)$ **until** $count = 0$;

          $sum_1 := 0$;

          for $index := N$ **downto** $K + 1$ **do**

              for $i := 1$ **to** $index$ **do**

                  $sum_1 := sum_1 + NumOnes(B_{index}[i], rowsize(i))$

              od

          od;

          $sum_2 := 0$;

          for $i := 1$ **to** $r - 1$ **do**

              $sum_2 := sum_2 + NumOnes(B_K[i], rowsize(i))$

          od;

          $sum_3 := 0$;

          for $j := 1$ **to** $val - 1$ **do**

              $sum_3 := sum_3 + B_K[r, j]$

          od;

          $id := sum_1 + sum_2 + sum_3$

        end $AssignId$

Figure 7: Procedure $AssignId$

given in the previous section used only bounded atomic registers. Therefore the three implementations outlined here provide means for symmetry breaking under three different assumptions of atomicity. In each case, our proof obligation is to verify that the implementation of $Initialize, Choose$, and $Scan$ terminate and satisfy the following two requirements:

Requirement ($SUC1$):

$useful\text{-}scan(S) \ \wedge \ [(\forall p, q : p, q \in active(K) : rand_p[i] \neq rand_q[i])$ immediately prior to $S] \ \Rightarrow$ $succ\text{-}scan(S)$

Requirement ($SUC2$):

For any successful scan S on the $i-$th row of A_K, $(\forall p, q : p, q \in active(K) \setminus choosing\text{-}set(S) : rand_p[i] \neq rand_q[i])$ immediately prior to S.

5.1 Implementation using Composite Registers

Implementation of the three procedures $Initialize, Choose$, and $Scan$ is straightforward in a system model permitting composite registers. Each row of array A_K is implemented by a composite register that is $rowsize(K) \geq K$ bits long. Procedure $Initialize(K, i)$ is implemented by setting the composite register $A_K[i]$ of array A_K to zeros. Procedure $Choose(K, i, l)$ is implemented by setting bit l of composite register $A_K[i]$ to zero, choosing a new random value in l, and setting bit l in $A_K[i]$ to one. Operation $Scan(K, i, success)$ is implemented by a single atomic read of the composite register $A_K[i]$ and setting $success$ to true if and only if K ones are observed. The procedures are shown in Figure 8. Requirements ($SUC1$) and ($SUC2$) are satisfied trivially as the entire contents of $A_K[i]$ are read in a single atomic step.

5.2 Implementation using Unbounded Variables

In this section we weaken the underlying system model to one that does not permit composite registers but allows unbounded atomic variables. In the implementation, all processes maintain a private timestamp cts that is incremented with each choose operation. This timestamp is affixed to each value written so that a scan operation can distinguish between two choices by the same process.

Type $rowtype$ consists of an array of $rowsize(K) \geq K$ records. Each record is an atomic variable consisting of a val bit and an integer component ts. A read or a write action on a record accesses both fields of the record. This is unlike a composite register, where the grain size of read actions is larger than that for write actions.

Procedure $Initialize$ resets cts and all the fields of array $A_K[i]$ to zeros. Procedure $Choose(K, i, l)$ is implemented by resetting previous choices in record $A_K[i][l]$ to zeros, choosing a new random value in l, incrementing the private timestamp, and setting $A_K[i][l]$ to $(1, cts)$. Procedure $Scan$ is implemented by first reading all the timestamps in array $A_K[i]$ that have val set to 1 and their positions. If the number of such timestamps is different from K then $success$ is set to false. Otherwise, the K timestamps and the associated positions are stored in a private array X. The tuples in array X are then sorted in a decreasing order of timestamps by procedure $Sort$. Next, array $A_K[i]$ is read once more according to the order of positions found in the sorted array X. If the newly read entries are identical to the previously stored entries in X (i.e., val bit is set to one and the timestamp has not changed) then $success$ is set to true. Otherwise, $success$ is set to false.

```
Type
    rowtype(K) = composite register [1 ‥ rowsize(K)] of 0|1;

Procedure Initialize(K, i):
    var  private j : integer;
    begin
        for  j := 1 to rowsize(K) do A_K[i][j] := 0 od
    end Initialize


Procedure Choose(K, i, l):
    begin
        A_K[i][l] := 0;
        l := Random(1 ··· rowsize(K));
        A_K[i][l] := 1
    end Choose


Procedure Scan(K, i, success):
    var  private X : array[1 ‥ rowsize(K)] of 0|1;
    begin
        X := A_K[i];
        if  NumOnes(X, rowsize(K)) = K then  success := true
        else  success := false fi
    end Scan
```

Figure 8: Implementation using Composite Registers: Procedures *Initialize*, *Choose*, and *Scan*.

```
Type
    rowtype(K) = array[1 .. rowsize(K)]  of record
                                             val : 0|1;
                                             ts : integer
                                           end record;
Procedure Initialize(K, i):
    var  private j : integer;
    begin
        cts := 0;
        for  j := 1 to rowsize(K) do A_K[i][j] := (0, 0) od
    end Initialize

Procedure Choose(K, i, l):
    begin
        A_K[i][l] := (0, 0);
        l := Random(1 ··· rowsize(K));
        cts := cts + 1;
        A_K[i][l] := (1, cts)
    end Choose

Procedure Scan(K, i, success):
    var  private X : array[1 .. rowsize(K)] of record
                                               ts : integer;
                                               pos : integer
                                             end record;
          j, m, ones, ts : integer;
          b : boolean;
    begin
        j, ones := 1, 0;
        while  j ≤ rowsize(K) do
            (b, ts) := A_K[i][j];
            if  b  then  ones := ones + 1 ; X[ones] := (ts, j) fi;
            j := j + 1
        od;
        if ones = K then
            Sort(X);
            j, success := 1, true;
            while  j ≤ K  ∧  success do
                (ts, m) := X[j];
                if A_K[i][m] = (1, ts) then j := j + 1
                else success := false fi
            od
        else success := false
        fi
    end Scan
```

Figure 9: Implementation using Unbounded Variables: Procedures *Initialize*, *Choose*, and *Scan*.

In order to prove requirement ($SUC1$), consider a useful scan operation S for which all K processes have set distinct entries in $A_K[i]$. Since S is useful, no process will write to $A_K[i]$ during the execution of S. Therefore S will find $ones = K$ in Figure 9. The absence of any writes to $A_K[i]$ during S implies that that no entry in $A_K[i]$ will be altered, and thus all comparisons in the second **while** loop will succeed. S will therefore set *success* to true.

In order to prove requirement ($SUC2$), let S be a successful scan operation performed by a process p. It is possible to show that during the execution of the second **while** loop of procedure *Scan* in Figure 9, process p reads at most one record written by the same process. The proof is based on the fact that a process increments its private timestamp between any two choose operations and process p sorts array X before reading the entries again. From procedure *Scan* in Figure 9, a successful scan reads K ones in array $A_K[i]$ during the execution of the second **while** loop. By the property proved above, all of these K ones must have been written by distinct processes. It follows that requirement ($SUC2$) is satisfied.

5.3 Implementation using Bounded Variables

Here we detail the implementation of procedures *Initialize*, *Choose*, and *Scan* in the weakest of the three models, i.e., a model allowing only *boolean* atomic variables. As shown in Figure 10, in this implementation each row $A_K[i]$ is composed of two arrays. Array *val* is used to record the random choices by the processes and array *flag* is used to identify when a scan operation is useful.

```
Type
  rowtype(K) = record
                 val : array[1 .. rowsize(K)] of 0|1;
                 flag : array[1 .. K] of 0|1;
               end record
```

Figure 10: Record *rowtype*

Procedures $Initialize(K, i)$ and $Choose(K, i, l)$ are shown in Figure 11. *Initialize* simply resets all the bits in $A_K[i]$ to zeros. Procedure $Choose(K, i, l)$ first resets the previous choice in the l-th bit of $A_K[i].val$ and all *flag* bits in $A_K[i]$ to zero. It then chooses a new random value for l, sets the l-th bit of $A_K[i].val$ to one, and returns l to the calling procedure.

Procedure $Scan(K, i, success)$ scans array $A_K[i].val$ of row $A_K[i]$ for K ones. It is implemented with the help of the recursive procedure $Rscan(n, K, i, success)$ shown in Figure 12. The predicate $n \leq K$ holds for any call of the recursive procedure $Rscan(n, K, i, success)$, and $Rscan(K, K, i, success)$ defines the implementation of $Scan(K, i, success)$. When *Rscan* is invoked with n in the range $2 < n \leq K$ as the first argument, it invokes *Rscan* with first argument $n - 1$. If the latter invocation results in *success* being set to true, then $A_K[i].flag[n]$ is set to one. By construction, each *Choose* and *Initialize* operation resets all *flag* bits to zero. Consequently, observing $A_K[i].flag[n] = 0$ at a later state indicates an interference by a *Choose* or an *Initialize* operation. Each iteration of the **while** loop in *Rscan* therefore checks bit $A_K[i].flag[n]$, and also checks whether $A_K[i].val$ contains K ones. $Rscan(n, K, i, success)$ sets *success* to true if and only if it reads K ones and finds the n-th flag bit to be one in each of the K iterations. Performing K iterations of this loop allows us to apply the pigeonhole principle on the number of iterations and the number of interfering processes. The exact details can be found in the proof of Proposition 6 that is stated at the end of this section. When

```
Procedure Initialize(K, i):
   var  private j : integer;
   begin
         for j := 1 to K do A_K[i].flag[j] := 0 od;
         for j := 1 to rowsize(K) do A_K[i].val[j] := 0 od
   end Initialize

Procedure Choose(K, i, l):
   var  private j : integer;
   begin
         A_K[i].val[l] := 0;
         for j := 1 to K do A_K[i].flag[j] := 0 od;
         l := Random(1 ··· rowsize(K));
         A_K[i].val[l] := 1
   end Choose
```

Figure 11: Implementation using bounded variables: Procedures *Initialize* and *Choose*

Rscan is invoked with first argument $n \leq 2$, there is no need of any recursive calls or the setting of any *flag* bits. The number of ones in $A_K[i].val$ is counted and *success* is set to true if K ones are read.

In order to prove that the implementation satisfies requirements ($SUC1$) and ($SUC2$), we need Propositions 5 and 6 stated below. The proofs of these propositions are given in the appendix. First, we introduce the following notation. For any invocation R of *Rscan* on a row of array A_K, let *choosing-set*(R) denote the set of processes that are executing a choose operation during the execution of R. Invocation R is said to be *useful* if *comp-init*(K) holds immediately prior to the execution of R, and if *choosing-set*(R) is empty.

Proposition 5 *Any* useful *invocation R of Rscan$(n, K, i, success), n \leq K$, for which $(\forall p, q \in active(K) : rand_p[i] \neq rand_q[i])$ holds immediately prior to R, sets success to true.*

Proposition 6 *For any execution R of Rscan$(n, K, i, success), 2 \leq n \leq K$, that sets success to true, $|choosing\text{-}set(R)| \leq n - 2 \;\Rightarrow$ $(\forall p, q : p, q \in active(K) \setminus choosing\text{-}set(R) : rand_p[i] \neq rand_q[i])$ at the state immediately prior to R.*

Requirement ($SUC1$) follows from Proposition 5 by taking $n := K$ and noting that $Scan(K, i, success)$ is implemented by $Rscan(K, K, i, success)$.

In order to prove ($SUC2$), first assume $K \geq 2$. If $|choosing\text{-}set(S)| \leq K - 2$, then the consequent of ($SUC2$) follows from Proposition 6 by substituting $n := K$ and noting that a scan operation on row $A_K[i]$ is implemented by invoking $Rscan(K, K, i, success)$. If $|choosing\text{-}set(S)| > K - 2$, then from Proposition 1, $|active(K) \setminus choosing\text{-}set(S)| \leq 1$, and the consequent holds vacuously. Now, consider the other case, $K < 2$. In this case, from Proposition 1, $|active(K)| \leq 1$. Therefore the consequent of ($SUC2$) again holds vacuously. This completes the proof of requirement ($SUC2$).

```
Procedure Rscan(n, K, i, success):
    var private j : integer;
    begin
        if n > 2 then
            Rscan(n − 1, i, K, success);
            if success then
                j := 1;
                A_K[i].flag[n] := 1;
                while j ≤ K and success do
                    if (NumOnes(A_K[i].val, rowsize(K)) = K) ∧ (A_K[i].flag[n] = 1)
                    then j := j + 1 else success := false fi
                od
            fi
        else
            if NumOnes(A_K[i].val, rowsize(K)) = K then success := true
            else success := false fi
        fi
    end Rscan
```

Figure 12: Implementation using bounded variables: Procedure *Rscan*

6 Time and Space Requirements

6.1 Synchronous Case

Before we analyze the complexity of the asynchronous algorithm, we consider the related problem of minimizing the expected running time of the synchronous algorithm in Figure 2. If the array X has $x \geq N$ bits, then in an execution of the **repeat** loop in Figure 2 by N processes, the probability that each process picks a different bit is

$$p_N(x) = \frac{x(x-1)\cdots(x-N+1)}{x^N} \quad .$$

If we assume that procedure *Random* takes constant time, then each scan operation on X requires $O(x)$ operations. Consequently the expected termination time of the synchronous algorithm is

$$O\left(\sum_{k\geq 1} k \cdot x p_N(x)(1-p_N(x))^{k-1} \right) \;=\; O\left(\frac{x}{p_N(x)}\right) \; .$$

Therefore given N, the problem is to determine the value $x_N \geq N$ that minimizes $x/p_N(x)$. The proof of the following result can be found in the appendix.

Proposition 7 *Let x_N be the value of $x \geq N$ that minimizes $E_N(x) = x/p_N(x)$. Then*

1. $\lim_{N\to\infty} x_N \;=\; \frac{1}{2}N^2$,

2. $\lim_{N\to\infty} E_N(x_N)/x_N = e$ *where $e = 2.718..$ is the base of the natural logarithm.*

In particular, if the shared memory array X is taken to be about $\frac{1}{2}N^2$ bits long, then the expected termination time $E_N(x_N)$ of the synchronous algorithm in Figure 2 is $O(N^2)$.

6.2 Asynchronous Case

In this section we analyze the time and space requirements of our asynchronous algorithm for each of three system models. We analyze the time required by our algorithm in terms of its round complexity [14]. A round is a minimal time interval over which every process which has not yet terminated executes at least one step. The round complexity of of an asynchronous algorithm for a given input is the maximum number of rounds over all possible computations for that input. Our initial time analysis is parameterized in terms of the complexity of the procedures *Initialize*, *Choose*, and *Scan*: we then specialize these quantities to each of the three models considered. To this end let $T_I(K)$, $T_C(K)$, and $T_S(K)$ denote the time complexity of *Initialize*, *Choose* and *Scan* operations on a single row of array A_K, respectively. For convenience, we assume that $rowsize(K)$ and $T_I(K)$ are increasing functions of K.

Let $\delta(K)$ denote the number of rounds required for a process to perform one iteration of the outer **repeat** loop of procedure *Toss* in Figure 3. It is easy to show that the number of rounds required by procedure *Detect* is $O(N^2 \cdot rowsize(N))$. Consequently,

$$\delta(K) = O(K \cdot (T_C(K) + T_S(K)) + N^2 \cdot rowsize(N)). \qquad (1)$$

The first part of the following proposition gives the expected number of rounds for the conclusion of Theorem 3 to hold and the second part defines a similar bound for Proposition 4. The proof is given in the appendix. Here p_K denotes the probability that a useful scan on a row of array A_K is successful.

Proposition 8

1. *Starting from any state in which* $|active(K)| = K$, *a state in which* $|successful(K)| > 0$ *is reached within*

$$O(K \cdot T_I(K) + \frac{\delta(K)}{p_K})$$

   expected number of rounds.

2. *Starting from any state in which* $p \in active(K) \ \wedge \ |SUCCESSFUL| = L$, *a state in which*

$$p \in SUCCESSFUL \ \vee \ (p \in active(N-L-1) \ \wedge \ |SUCCESSFUL| > L\ )$$

   is reached within

$$O(\frac{\delta(N-L)}{p_{N-L}} + (N-L)T_I(N-L) + \sum_{i=N-L}^{K}(1+\delta(i)))$$

   expected number of rounds.

Based on this proposition, we prove

Theorem 6 *The expected number of rounds required by the asynchronous algorithm in Figure 6 is*

$$O(N^2 rowsize(N) + N^2 T_I(N) + \sum_{K=1}^{N} \frac{\delta(K)}{p_K}) \qquad (2)$$

Proof First we compute the expected number of rounds required for the conclusion of Theorem 4 to hold. Within $N^2 \cdot rowsize(N)$ rounds all processes complete the initialization of B arrays and begin executing $Toss(N)$. Consider the sequence of states $s_0, s_1, \ldots$ and the increasing sequence $m_0, m_1, \ldots$ considered in the proof of Theorem 4. The expected number of rounds between states s_j and s_{j+1} is given by part 2 of Proposition 8 as

$$O(\frac{\delta(N-m_{j+1})}{p_{N-m_{j+1}}} + (N-m_{j+1})T_I(N-m_{j+1}) + \sum_{i=N-m_{j+1}}^{N} (1+\delta(i))) \ .$$

Summing this quantity over all the consecutive states in the sequence, the last term above becomes

$$\sum_{K=1}^{N} (1+\delta(K))$$

regardless of the values of the m_j, and the sum of the first two terms is

$$O(\sum_{K=1}^{N} \frac{\delta(K)}{p_K} + \sum_{K=1}^{N} K \cdot T_I(K)).$$

Using the fact that $\sum_{K=1}^{N} K \cdot T_I(K) = O(N^2 T_I(N))$ and also including the complexity due to initialization, the expected number of rounds for every process to perform a successful scan is

$$O(N^2 rowsize(N) + N^2 T_I(N) + \sum_{K=1}^{N} \frac{\delta(K)}{p_K} + \sum_{K=1}^{N} (1+\delta(K))) \ ,$$

which can be rewritten as (2).

Now we consider the time complexity for the generation of unique ids once all processes have become successful. Within $O(1)$ rounds processes copy their choices into B arrays and in an additional $O(N^2 rowsize(N))$ rounds all processes detect *count* $= 0$ and call procedure $AssignId$. Within another $O(1) + O(N^2 rowsize(N))$ rounds all processes detect *count* $= 0$ in procedure $AssignId$ and the lexicographic ordering starts. The computation of sum_1, sum_2, sum_3 and *id* then completes in another $O(N^2 rowsize(N))$ rounds. Therefore the generation of unique ids after every process is successful requires $O(N^2 rowsize(N))$ rounds. Therefore (2) is the complexity of the algorithm as stated. □

Now we can use the expression (1) for $\delta(K)$ and expression (2) for the expected number of rounds of the algorithm and specialize the implementation dependent quantities $T_I(N)$, $T_C(K)$, and $T_S(K)$, for composite variables, unbounded atomic variables, and bounded atomic variables separately. We assume that function *Random* takes constant amount of time.

6.2.1 Composite Variables

In this case

$$\begin{aligned} T_I(K) &= O(rowsize(K)), \\ T_C(K) &= O(1), \\ T_S(K) &= O(rowsize(K)) \ , \end{aligned}$$

and therefore from (1),

$$\delta(K) = O(N^2 rowsize(N)).$$

From (2), the expected number of rounds is

$$O(\ N^2 rowsize(N) + N^2 rowsize(N) + \sum_{K=1}^{N} \frac{N^2 rowsize(N)}{p_K}\ )\ =\ O(N^2 rowsize(N) \sum_{K=1}^{N} \frac{1}{p_K})\ .$$

The space requirements in this case are $O(N^2 rowsize(N))$ atomic shared variables that are used for the arrays B_K and $O(N^2)$ composite variables used for the arrays A_K.

6.2.2 Unbounded Atomic Variables

In this case

$$\begin{aligned} T_I(K) &= O(rowsize(K)), \\ T_C(K) &= O(1), \\ T_S(K) &= O(K \log K + rowsize(K)). \end{aligned}$$

Therefore

$$\delta(K) = O(N^2 rowsize(N)).$$

The expected number of rounds is found from (2) as

$$O(\ N^2 rowsize(N) + N^2 rowsize(N) + \sum_{K=1}^{N} \frac{N^2 rowsize(N)}{p_K}\ )\ =\ O(N^2 rowsize(N) \sum_{K=1}^{N} \frac{1}{p_K})\ .$$

The space requirements are $O(N^2 rowsize(N))$ bounded shared variables for the arrays B_K and $O(N^2 rowsize(N))$ unbounded atomic variables for the arrays A_K.

6.2.3 Bounded Atomic Variables

For bounded atomic variables,

$$\begin{aligned} T_I(K) &= O(rowsize(K)), \\ T_C(K) &= O(K), \\ T_S(K) &= O(K^2 rowsize(K)). \end{aligned}$$

The derivation of the expression for $T_S(K)$ is based on the recursive nature of procedure *Rscan*. Let $f(n, K)$ denote the number of rounds required for $Rscan(n, K, i, success)$ to complete. Then for $n \leq 2$, $f(n, K) = O(rowsize(K))$, and for $n > 2$, $f(n, K)$ satisfies

$$f(n, K) = f(n-1, K) + O(K \cdot rowsize(K))\ .$$

It follows that $f(n, K) = O(n \cdot K \cdot rowsize(K))$. Since $Scan(K, i, success)$ is implemented by invoking $Rscan(K, K, i, success)$, we find $T_S(K) = O(K^2 \cdot rowsize(K))$. Now from (1)

$$\delta(K) = O(N^3 rowsize(N)).$$

The expected number of rounds in this case is found to be

$$O(N^2 rowsize(N) + N^2 rowsize(N) + \sum_{K=1}^{N} \frac{N^3 rowsize(N)}{p_K} \ ) \ = \ O(N^3 rowsize(N) \sum_{K=1}^{N} \frac{1}{p_K} \ ) \ .$$

The space requirements are $O(N^2 rowsize(N))$ boolean atomic variables that are used for the arrays A_K and B_K.

From part 2 of Proposition 7 we have that if $rowsize(N) = \frac{1}{2}N^2$, then $p_N(rowsize(N))$ approaches e^{-1}. Therefore by taking $rowsize(K) := \frac{1}{2}K^2$,

$$rowsize(N) \sum_{K=1}^{N} \frac{1}{p_K} \ = \ O(\, N^3 \,).$$

Consequently the expected number of rounds for the completion of $Main$ assuming composite or unbounded variables is $O(\, N^5 \,)$. For bounded atomic variables, the expected number of rounds is $O(\, N^6 \,)$.

SHARED VARIABLE MODEL	EXPECTED NO. ROUNDS	SPACE REQUIREMENTS
Composite Registers	$O(N^5)$	$O(N^2)$ composite $O(N^4)$ bounded atomic
Unbounded Atomic Variables	$O(N^5)$	$O(N^4)$ unbounded atomic $O(N^4)$ bounded atomic
Bounded Atomic Variables	$O(N^6)$	$O(N^4)$ bounded atomic

Figure 13: Time and Space Requirements.

7 Concluding Remarks

In this paper we constructed a hierarchical solution to the problem of symmetry breaking in an asynchronous system using shared variables. First we determined a number of properties such a solution must have: we showed that the exact number of processes must necessarily be known to the processes in order to solve the problem. We also showed that termination requires participation by every process. This means that failures cannot be tolerated during symmetry breaking unless one assumes other means for detecting failures.

Generalizing from a randomized synchronous solution to the problem, we presented procedures $Main$ and $Toss$, and derived specifications $SUC1, SUC2, DET1$, and $DET2$ for four procedures $Initialize$, $Choose$, $Scan$, and $Detect$ that form the basic building blocks of the general asynchronous algorithm. The first three of these procedures used a sequence of shared arrays A_K whose rows were of an unspecified atomicity. We argued the correctness of $Main$ assuming an implementation of $Initialize$, $Choose$, $Scan$, and $Detect$ in which the four specifications set forth in the proof of $Main$ were satisfied. Then we presented an implementation of $Detect$, which used another sequence of shared arrays B_K consisting of boolean atomic variables. Subsequently, we showed how distinct ids could be assigned using procedure $AssignId$.

In Section 5, we considered the implementation of procedures *Initialize*, *Choose*, and *Scan*, and of each row of array A_K under three different atomicity assumptions: composite variables, unbounded atomic variables, and bounded atomic variables. The implementations became more difficult and required increasing number of rounds as the atomicity assumptions became more stringent. For example, the implementation of *Scan* required $O(rowsize(N))$ steps under the composite variable model, $O(N \log N + rowsize(N))$ steps under the unbounded atomic variable model, and $O(N^2 rowsize(N))$ steps under the bounded atomic variable model. However, we showed in Section 6 that the running time of the entire algorithm is governed by the implementation of *Detect* for composite variables and unbounded atomic variables, and by the implementation of *Scan* for bounded atomic variables. The time and space requirements for these three cases are summarized in Figure 13. It may be possible to implement procedure *Detect* more efficiently by associating a counter with each B array; however, this would not have a bearing on the final implementation using boolean atomic variables.

We did not assume initial values for shared variables as such an assumption presupposes a distinct process that carries out the needed initializations. Our solutions were made more difficult on this account. Lack of initial values is the reason that in *Main*, the recording of random choices by a successful process required repeated copy operations into arrays B_K, and procedure *AssignId* used a form of a barrier synchronization via the variable *count*.

Note that the usual hierarchy of register constructions from boolean safe registers to multi-valued atomic registers (and composite registers) [22] assumes process ids. Therefore, the kind of primitives assumed in order to solve the symmetry breaking problem becomes crucial, since existing register constructions cannot be used to weaken the primitives later. Our final algorithm of Section 5.3 uses boolean atomic variables. Recently, Kutten, Ostrovsky, and Patt-Shamir have solved the same problem assuming atomic variables of size $O(\log N)$ [20]. The two solutions are not equivalent because the implementation of larger atomic shared variables from smaller ones seems to require unique process ids. The solvability of the symmetry breaking problem with primitives weaker than atomic boolean variables remains open.

References

[1] Abrahamson, K., On Achieving Consensus Using a Shared Memory, Proceedings of the 7th ACM Symposium on Principles of Distributed Computing, 1988, pp. 291–302.

[2] Afek, Y., H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, Atomic Snapshots of Shared Memory, Proceedings of the 9th ACM Symposium on Principles of Distributed Computing, 1990, pp. 1–14.

[3] Anderson, J. H., and M. G. Gouda, The Virtue of Patience: Concurrent Programming with and without Waiting, Technical Report TR.90.23, Department of Computer Sciences, University of Texas at Austin, July 1990.

[4] Anderson, J. H., Composite Registers, Proceedings of the 9th ACM Symposium on Principles of Distributed Computing, 1990, pp. 15–30.

[5] Angluin, D., Local and Global Properties in Networks of Processors, Proceedings of the 12th ACM Symposium on Theory of Computing, 1980, pp. 82–93.

[6] Aspnes, J., and M. Herlihy, Fast Randomized Consensus Using Shared Memory, Journal of Algorithms, 11(3), 1990, pp. 441–461.

[7] Attiya, H., A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk, Achievable Cases in an Asynchronous Environment, Proceedings of the 28th Symposium on Foundations of Computer Science, 1987, pp. 337–346.

[8] Attiya, H., D. Dolev, and N. Shavit, Bounded Polynomial Randomized Consensus, 8th ACM Symposium on Principles of Distributed Computing, 1989, pp. 281–294.

[9] Bar-Noy, A., M. Ben-Or, and D. Dolev, Choice Coordination with Bounded Failure, Distributed Computing, 3(2), 1989, pp. 61–72.

[10] Burns, J. E., Symmetry in Systems of Asynchronous Processes, Proceedings of the 22nd Symposium on Foundations of Computer Science, 1981, pp. 169–174.

[11] Chandy, K. M., and J. Misra, The Drinking Philosophers Problem, ACM Transactions on Programming Languages and Systems, 6(4), 1984, pp. 632–646.

[12] Chandy, K. M., and J. Misra, *Parallel Program Design: A Foundation*, Addison Wesley, 1988.

[13] Chor, B., Israeli, A., and Li, M., On Processor Coordination Using Asynchronous Hardware, Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, 1987.

[14] Cole, R., and O. Zajicek, The APRAM: Incorporating Asynchrony into the PRAM Model, Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures, 1989, pp. 169–178.

[15] Dijkstra, E. W., Solution of a Problem in Concurrent Programming Control, Communications of the ACM, 8(9), 1965, pp. 569.

[16] Dijkstra, E. W., Hierarchical Ordering of Sequential Processes, Acta Informatica 1, 1971, pp. 115–138.

[17] Herlihy, M., Wait-free Synchronization, ACM Transactions on Programming Languages and Systems, 13(1), 1991, pp. 124–149.

[18] Itai, A., and M. Rodeh, Symmetric Breaking in Distributed Networks, Proceedings of the 22nd Annual Symposium on Foundations of Computer Science, 1981, pp. 150–158.

[19] Johnson, R. E., and F. B. Schneider, Symmetry and Similarity in Distributed Systems, Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing, 1985, pp. 13–22.

[20] Kutten, S., R. Ostrovsky, and B. Patt-Shamir, The Las-Vegas Processor Identity Problem, Proceedings of Israel Symposium on Theory of Computing and Systems, 1993.

[21] Lamport, L., Time, Clock, and the Ordering of Events in a Distributed System, Communications of the ACM, 21(7), 1978, pp. 558 – 565.

[22] Lamport, L., On Interprocess Communication, parts I and II, Distributed Computing, 1(2), 1986, pp.77–101.

[23] Lehmann, D., and M. O. Rabin, On Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem, Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages, 1981, 122–138.

[24] Lim, L., and A. Park, Solving the Processor Identity Problem in $O(n)$ Space, Second IEEE Symposium on Parallel and Distributed Computing, 1990.

[25] Lipton, R. J., and A. Park, The Processor Identity Problem, Information Processing Letters, 36, 1990, pp. 91–94.

[26] Matias, Y., and Y. Afek, Simple and Efficient Election Algorithms for Anonymous Networks, Proceedings of the 3rd International Workshop on Distributed Algorithms, 1989, pp. 183–194.

[27] Pnueli, A., and Z. Manna, How to Cook a Temporal Proof System for Your Pet Language, Proceedings of the 9th ACM Symposium on Principles of Programming Languages, 1983, 141–154.

[28] Rabin, M. O., The Choice Coordination Problem, Acta Informatica, 17, 1982, pp. 121–134.

[29] Scheiber, B., and M. Snir, Calling Names on Nameless Networks, Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing, 1989, pp. 319–328.

[30] Teng, S., Space Efficient Processor Identity Protocol, Information Processing Letters, 34, 1990, pp. 147–154.

[31] Yamashita, M., and T. Kameda, Computing on Anonymous Networks, Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing, 1988, pp. 117–130.

8 Appendix

Proposition 1 $|active(K)| \leq K$.

Proof The proof is by induction on K. The base case $K := N$ follows trivially as the total number of processes is at most N. For the induction step, we prove it for $K-1$ by assuming it for $N, N-1 \ldots, K$. Assume, for the sake of contradiction, that

$|active(K-1)| > K-1$

holds at some state s. From the induction hypothesis,

$|active(K)| \leq K$

at state s. By the construction of procedure $Main$ in Figure 3,

$active(K-1) \subseteq active(K)$

at each state. Therefore, it follows that at state s,

$active(K-1) = active(K)$ and $|active(K-1)| = K$.

Again from the construction of procedure $Main$ in Figure 3, it follows that $|done(K)| = K$ at state s. Of all the K processes in $done(K)$ at state s, consider the first process p to enter $done(K)$. We will show next that process p must have executed a successful scan and consequently, contrary to our assumption, $|successful(K)| > 0$ and $|active(K-1)| \leq K-1$. Consider any invocation of $Detect$ by process p before joining $done(K)$. Since $|SUCCESSFUL| \leq N-K$ at state s, by requirement $(DET1)$, process p finds $count_p \geq N-(N-K)$, i.e., $count_p \geq K$. Therefore, process p cannot exit on account of finding $count_p < K$. Thus, p must exit on account of performing a successful scan. The proof of Proposition 1 follows. □

Lemma 1 $|active(K)| = K \ \wedge \ |done(K)| = 0$ *unless* $(\exists S :: S$ *is a successful scan on* $A_K)$.

Proof Consider any state s at which $|active(K)| = K \ \wedge \ |done(K)| = 0$ holds. We will show that at the next state t, either $|active(K)| = K \ \wedge \ |done(K)| = 0$ or a successful scan has occurred on some row of array A_K. By definition, membership in $active(K)$ is stable. Therefore, $|active(K)| \geq K$ at state t. Applying Proposition 1, $|active(K)| = K$ at state t. Next, we assume that $|done(K)| > 0$ at state t and show that some process has performed a successful operation by state t. Let p be the process that belongs to $done(K)$ at state t. This implies that process p either exited on account of executing a successful scan or on account of finding $count_p < K$. We next show that the latter case is not possible, thus meeting the proof obligation. Since $|active(K)| = K \ \wedge \ |done(K)| = 0$ at state s, $|SUCCESSFUL| \leq N-K$ at state s. Consider any invocation of $Detect$ by process p. Since $|SUCCESSFUL| \leq N-K$ at state s, by requirement $(DET1)$, process p finds $count_p \geq N-(N-K)$, i.e., $count_p \geq K$. Therefore, process p cannot exit on account of finding $count_p < K$. The proof of Lemma 1 follows. □

Proposition 2 $|active(K)| = K \ \wedge \ |done(K)| > 0 \ \Rightarrow \ (\exists S :: S$ is a successful scan on $A_K)$.

Proof Note that the above predicate is true initially as $|done(K)| = 0$ initially. Therefore, it suffices to show that the predicate is stable. Since the consequent of the predicate is stable, the proof will follow if we show that

$\neg(|active(K)| = K \ \wedge \ |done(K)| > 0)$ *unless* $(\exists S :: S$ is a successful scan on $A_K)$.

Rewriting the antecedent using Proposition 1,

$(|active(K)| < K \ \wedge \ |done(K)| > 0) \ \vee \ (|active(K)| < K \ \wedge \ |done(K)| = 0) \ \vee$
$(|active(K)| = K \ \wedge \ |done(K)| = 0)$ *unless* $(\exists S :: S$ is a successful scan on $A_K)$.

In the remainder of the proof we consider each disjunct in the antecedent separately and show that

1. $|active(K)| < K \ \wedge \ |done(K)| > 0$ *unless* $(\exists S :: \ S$ is a successful scan on $A_K)$,

2. $|active(K)| < K \ \wedge \ |done(K)| = 0$ *unless*
   $(|active(K)| < K \ \wedge \ |done(K)| > 0) \ \vee \ (|active(K)| = K \ \wedge \ |done(K)| = 0)$, and

3. $|active(K)| = K \ \wedge \ |done(K)| = 0$ *unless* $(\exists S :: \ S$ is a successful scan on $A_K)$.

The desired property then follows by taking the disjunction of the above three properties.

Proof of 1:
Consider any state s at which $(|active(K)| < K \ \wedge \ |done(K)| > 0)$ holds. We will show that at the next state t, either $(|active(K)| < K \ \wedge \ |done(K)| > 0)$ or a successful scan has occurred on some row of array A_K. By definition, membership in $done(K)$ is stable. Therefore, $|done(K)| > 0$ at state t. Next, we assume $\neg(|active(K)| < K)$ at state t, and prove that some process has performed a successful operation by state t. By Proposition 1, $|active(K)| = K$ at state t. Let p be the first process to join $done(K)$ at some state u. Process p either exited on account of executing a successful scan or on account of finding $count_p < K$. We next show that the latter case is not possible, thus meeting the proof obligation. Since $|active(K)| = K$ at state t and process p is the first process to join $done(K)$, $|SUCCESSFUL| \leq N - K$ before state u. Consider any invocation of *Detect* by process p. Since $|SUCCESSFUL| \leq N - K$ during this invocation, by requirement $(DET1)$, process p finds $count_p \geq N - (N - K)$, i.e., $count_p \geq K$. Therefore, process p cannot exit on account of finding $count_p < K$. The proof follows.

Proof of 2:
Consider any state s at which $|active(K)| < K \ \wedge \ |done(K)| = 0$ holds. We need to show at the next state t,
$(|active(K)| \leq K \ \wedge \ |done(K)| = 0) \ \vee \ (|active(K)| < K \ \wedge \ |done(K)| > 0)$.
Considering all the possibilities at state t,

(a) a process joins $done(K)$ and $active(K)$ remains unchanged, or

(b) a process joins $active(K)$ and $done(K)$ remains unchanged, or

(c) $active(K)$ and $done(K)$ remain unchanged.

In the former case, $(|active(K)| < K \ \wedge \ |done(K)| > 0)$ at the next state and in the latter two cases, $(|active(K)| \leq K \ \wedge \ |done(K)| = 0)$ at the next state. The proof follows.

Proof of 3: Follows from Lemma 1.
This concludes the proof of Proposition 2. □

Lemma 2 $|active(K)| = K \ \wedge \ |done(K)| = 0 \ \Rightarrow count_p \geq K$

Proof Consider any state s at which $|active(K)| = K \ \wedge \ |done(K)| = 0$. This implies that $|SUCCESSFUL| \leq N - K$ at state s. Consider any invocation of procedure *Detect* by process p that terminates and sets $count_p$ at or before state s. Since $|SUCCESSFUL| \leq N - K$ at state s, by requirement $(DET1)$, $count_p \geq N - (N - K)$. In other words, $count_p \geq K$. □

In the proofs of the following three lemmas, we will be concentrating on a fixed value of K and therefore, in order to keep the exposition simple, we will omit the parameter K in certain terms

(e.g., *active* and *comp-init*) when no confusion arises. Let $\#choices(p,i)$ denote the total number of completed executions of $Choose$ operations on row $A_K[i]$ by process p and let $\#useful$ denote the total number of useful scan operations executed by all processes in $active(K)$. Let p range over processes in the set $active(K)$ and let S range over scan operations on rows of A_K.

Lemma 3 $comp\text{-}init \ \wedge \ (\forall p,i :: \#choices(p,i) = x_{p,i}) \ \mapsto$
$(comp\text{-}init \ \wedge \ (\forall p,i :: \#choices(p,i) > x_{p,i})) \ \vee \ (\exists S :: succ\text{-}scan(S))$.

Proof Since the implementations of $Scan, Choose$, and $Detect$ terminate, and since any given process p in Figure 3 keeps on executing the repeat-until loops until a successful scan is performed or $count_p$ becomes less than K,

$comp\text{-}init_p \ \wedge \ (\forall i :: \#choices(p,i) = x_{p,i}) \ \mapsto$
$(\forall i :: \#choices(p,i) > x_{p,i}) \ \vee \ (\exists S :: succ\text{-}scan(S)) \ \vee \ count_p < K.$

From Lemma 1,

$|active| = K \ \wedge \ |done| = 0 \ unless \ (\exists S :: succ\text{-}scan(S)).$

Combining these two properties,

$comp\text{-}init_p \ \wedge \ |active| = K \ \wedge \ |done| = 0 \ \wedge \ (\forall i :: \#choices(p,i) = x_{p,i}) \ \mapsto$
$(\forall i :: \#choices(p,i) > x_{p,i}) \vee (\exists S :: succ\text{-}scan(S)) \vee (|active| = K \wedge |done| = 0 \wedge count_p < K).$

Using Lemma 2,

$comp\text{-}init_p \ \wedge \ |active| = K \ \wedge \ |done| = 0 \ \wedge \ (\forall i :: \#choices(p,i) = x_{p,i}) \ \mapsto$
$(\forall i :: \#choices(p,i) > x_{p,i}) \ \vee \ (\exists S :: succ\text{-}scan(S)).$

Taking the conjunction over all p and noting that the consequent is stable,

$(\forall p :: comp\text{-}init_p) \ \wedge \ |active| = K \ \wedge \ |done| = 0 \ \wedge \ (\forall p,i :: \#choices(p,i) = x_{p,i}) \ \mapsto$
$(\forall p,i :: \#choices(p,i) > x_{p,i}) \ \vee \ (\exists S :: succ\text{-}scan(S)).$

Since $(\forall p :: comp\text{-}init_p) \ \wedge \ |active| = K \ \equiv \ comp\text{-}init$,

$comp\text{-}init \ \wedge \ |active| = K \ \wedge \ |done| = 0 \ \wedge \ (\forall p,i :: \#choices(p,i) = x_{p,i}) \ \mapsto$
$(\forall p,i :: \#choices(p,i) > x_{p,i}) \ \vee \ (\exists S :: succ\text{-}scan(S)).$

From Proposition 2,

$|active| = K \ \wedge \ |done| > 0 \ \Rightarrow \ (\exists S :: succ\text{-}scan(S)).$

Combining the above two properties,

$comp\text{-}init \ \wedge \ |active| = K \ \wedge \ (\forall p,i :: \#choices(p,i) = x_{p,i}) \ \mapsto$
$(\forall p,i :: \#choices(p,i) > x_{p,i}) \ \vee \ (\exists S :: succ\text{-}scan(S)).$

Since $comp\text{-}init \ \Rightarrow |active| = K$,

$comp\text{-}init \ \wedge \ (\forall p,i :: \#choices(p,i) = x_{p,i}) \ \mapsto$
$(\forall p,i :: \#choices(p,i) > x_{p,i}) \ \vee \ (\exists S :: succ\text{-}scan(S)).$

The desired proof follows from the stability of *comp-init*. □

Lemma 4 $comp\text{-}init \ \wedge \ \#useful = y \ \mapsto (comp\text{-}init \ \wedge \ \#useful > y) \ \vee \ (\exists S :: succ\text{-}scan(S))$

Proof We first prove that in any complete execution of the inner repeat-until loop of Figure 3 by a process p, either there exists a successful scan, or the total number of useful scans increases. First we introduce some terminology. Consider the execution of the l-th iteration of the outer **repeat** loop of procedure $Toss(K)$ by some process p. We say that predicate $start(p,l)$ is true at a state if and only if process p has just completed its last choose operation during this iteration. Similarly, we say that predicate $end(p,l)$ is true at a state if and only if process p has just completed its last scan operation during this iteration. We prove the following lemma.

Lemma 5 $comp\text{-}init \ \wedge \ start(p,l) \ \wedge \ \#useful = y \ \mapsto$
$(end(p,l) \ \wedge \ (\#useful > y)) \ \vee \ (\exists S :: succ\text{-}scan(S)).$

Proof Let s be any state at which the predicate *comp-init* $\wedge$ $start(p,l)$ holds. Since the implementations of *Scan* and *Detect* terminate, there exists a later state t at which $end(p,l)$ holds. Let $CHOOSING\text{-}SET(p,l)$ denote the union of all $choosing\text{-}set(S)$, where S is a scan operation performed by process p between the states s and t. We prove the lemma by induction on the size of $CHOOSING\text{-}SET(p,l)$. If $|CHOOSING\text{-}SET(p,l)| = 0$ then each scan made by p is useful. Since p performs at least one scan between states s and t, the base case follows.

For the induction step, let $|CHOOSING\text{-}SET(p,l)| = m$ and assume the induction hypothesis for all sizes of the set less than m. If there exists a scan by any process that is successful, or a scan S by p for which $choosing\text{-}set(S) = \phi$ then the desired result follows. So, assume to the contrary that all scans by all processes are unsuccessful and that $choosing\text{-}set(S) \neq \phi$ for all scans S made by the process p between states s and t. From Figure 3 and Lemma 1, in the absence of any successful scans, process p performs K scan operations between states s and t. Let $S_i, 1 \leq i \leq K$, denote the scan operation made by process p on row $A_K[i]$. Since $p \notin CHOOSING\text{-}SET(p,l)$, and by Proposition 1, $active(K) \leq K$, we have $m < K$. Therefore, there must be some process $q \neq p$ and indices i, j with $i > j$ such that

$$q \in choosing\text{-}set(S_i) \wedge q \in choosing\text{-}set(S_j) \ .$$

Process p performs S_i on row $A_K[i]$, and later performs S_j on row $A_K[j]$. The predicate $choosing(q,i,K)$ is true at some state u and later the predicate $choosing(q,j,K)$ is true at some state v such that $s \mapsto u \mapsto v \mapsto t$. Now consider the behavior of q between the states u and v. Process q was executing a choose operation on row $A_K[i]$ at state u, and was later executing a choose operation on row $A_K[j]$ at state v. Since $i > j$, process q performs its choose operation on row $A_K[j]$ before its choose operation on row $A_K[i]$ during any iteration of the outer **repeat** loop. Therefore, states u and v occur during different iterations of the outer **repeat** loop of process q. Consequently, there exist n, n' with $n < n'$ such that process q was performing its n-th iteration of the outer **repeat** loop at state u and its n'-th iteration of the outer **repeat** loop at state v. It follows that there exist two states s' and t' such that $start(q,n)$ holds at s', $end(q,n)$ holds at t', and

$$s \ \mapsto \ u \ \mapsto \ s' \ \mapsto \ t' \ \mapsto \ v \ \mapsto \ t.$$

Since processes p and q do not belong to $choosing\text{-}set(S)$, where S is any scan performed by process q between the states s' and t', $|CHOOSING\text{-}SET(q,n)| \leq m-1$. Since *comp-init* is stable and it holds at state s, it also holds at state s'. Therefore, applying the induction hypothesis,

$$\#useful \text{ at state } s' < \#useful \text{ at state } t'.$$

Thus,

$$\#useful \text{ at state } s \leq \#useful \text{ at state } s' < \#useful \text{ at state } t' \leq \#useful \text{ at state } t.$$

The desired proof of the induction step follows. □

We now go back to the proof of Lemma 4. Since process p keeps on executing the repeat-until loops until a successful scan is performed or it finds $count_p < K$, and since the implementations of *Choose*, *Scan*, and *Detect* terminate,

$$comp\text{-}init \ \mapsto (\exists p, l :: start(p,l)) \ \vee \ (\exists S :: succ\text{-}scan(S)) \ \vee \ (\exists p :: count_p < K).$$

From Lemma 1,

$$|active| = K \ \wedge \ |done| = 0 \ unless \ \ (\exists S :: succ\text{-}scan(S)).$$

Combining the above two properties,

$$comp\text{-}init \ \wedge \ |active| = K \ \wedge \ |done| = 0 \ \mapsto$$
$$(\exists p, l :: start(p,l)) \ \vee \ (\exists S :: succ\text{-}scan(S)) \ \vee \ (\exists p :: count_p < K \ \wedge \ |active| = K \ \wedge \ |done| = 0).$$

Using Lemma 2,

$$comp\text{-}init \ \wedge \ |active| = K \ \wedge \ |done| = 0 \ \mapsto (\exists p, l :: start(p,l)) \ \vee \ (\exists S :: succ\text{-}scan(S)).$$

From Proposition 2,

$|active| = K \ \wedge \ |done| > 0 \ \Rightarrow \ (\exists S :: succ\text{-}scan(S))$.

Combining the above two properties,

$comp\text{-}init \ \wedge \ |active| = K \ \mapsto (\exists p, l :: start(p, l)) \ \vee \ (\exists S :: succ\text{-}scan(S))$.

Since $comp\text{-}init \ \Rightarrow |active| = K$,

$comp\text{-}init \ \mapsto (\exists p, l :: start(p, l)) \ \vee \ (\exists S :: succ\text{-}scan(S))$.

Noting that *comp-init* is stable,

$comp\text{-}init \ \mapsto (comp\text{-}init \ \wedge \ (\exists p, l :: start(p, l))) \ \vee \ (\exists S :: succ\text{-}scan(S))$.

Since $\#useful$ is monotonically increasing,

$comp\text{-}init \ \wedge \ \#useful = y \ \mapsto$

$(comp\text{-}init \ \wedge \ (\exists p, l :: start(p, l)) \ \wedge \ \#useful = y) \ \vee \ \#useful > y \ \vee \ (\exists S :: succ\text{-}scan(S))$.

Lemma 5 can be used to assert that

$(comp\text{-}init \ \wedge \ (\exists p, l :: start(p, l)) \ \wedge \ \#useful = y) \ \mapsto \#useful > y \ \vee \ (\exists S :: succ\text{-}scan(S))$.

Combining these two assertions,

$(comp\text{-}init \ \wedge \ \#useful = y) \ \mapsto \#useful > y \ \vee \ (\exists S :: succ\text{-}scan(S))$.

The proof of Lemma 4 follows from the stability of *comp-init*. □

Proposition 3 *Starting from any state in which comp-init*(K) *holds, either a successful scan occurs on some row of* A_K*, or there exists an infinite sequence* $S_m, m \geq 0$*, of useful scans by processes in active*(K) *such that*

1. *scan* S_m *is made on a row* r_m *of array* A_K*, and*
2. *each scan in the sequence observes the result of fresh random choices by the processes, i.e., for any two scans* S_{m_1} *and* S_{m_2} *with* $m_1 < m_2$*, if* $r_{m_1} = r_{m_2}$ *then each process in active*(K) *makes a fresh random choice on row* r_{m_1} *between scans* S_{m_1} *and* S_{m_2}.

Proof To prove the proposition, assume that there are no successful scans on rows of A_K. Consider any state s_0 at which the predicate *comp-init*(K) holds. Using Lemma 3, starting from s_0 eventually a state t_0 is reached in which every process has made a fresh choice on each row of A_K and *comp-init*(K) holds at t. Using Lemma 4, starting from state t_0 eventually a state s_1 is reached such that a useful scan S_0 has been made on some row of A_K between states t_0 and s_1 and *comp-init*(K) holds at s_1. Since *comp-init*(K) holds at s_1 and $s_0 \ \mapsto \ s_1$, the above argument can be repeated at state s_1. In this way, we generate an infinite sequence of scans satisfying the proposition.

□

Lemma 6 $|active(K)| > 0 \ \Rightarrow \ |\{p : p \in successful(L) \ \wedge \ L > K\}| \geq N - K$

Proof The proof is by induction on K. The base case $K := N$ holds trivially as the consequent is true. For the induction step, we assume the hypothesis for $K + 1, \ldots, N$, and prove it for K. Consider any state s at which $|active(K)| > 0$. By the construction of procedure $Main$ in Figure 3, $active(K) \subseteq done(K + 1) \subseteq active(K + 1)$. Therefore, $|active(K + 1)| > 0$ and $|done(K + 1)| > 0$ at state s. By the induction hypothesis,

$|\{p : p \in successful(L) \ \wedge \ L > K + 1\}| \geq N - K - 1$ at state s.

The proof follows if the left hand side in the above expression is greater than $N - K - 1$. So, let

$|\{p : p \in successful(L) \ \wedge \ L > K + 1\}| = N - K - 1$,

at state s. Now, consider the first process p that becomes a member of $done(K+1)$. Process p must have exited either on account of performing a successful scan or on account of finding $count_p < K + 1$. We will show next that the latter case is not possible. Consider any invocation of $Detect$ by process p before

joining $done(K+1)$. By assumption, $|SUCCESSFUL| \leq N-K-1$ during the invocation of $Detect$. Therefore, by requirement $(DET1)$, process p finds $count_p \geq N-(N-K-1)$, i.e., $count_p \geq K+1$ and consequently, cannot exit on account of finding $count_p < K+1$. Thus, $p \in successful(K+1)$, and

$|\{p : p \in successful(L) \ \wedge \ L > K\}|$
$= \ |\{p : p \in successful(L) \ \wedge \ L > K+1\}| + |\{p : p \in successful(K+1)\}|$
$\geq \ N-K-1+1.$

The desired proof follows. □

Lemma 7 $p \in active(K) \ \wedge \ (K > N - |SUCCESSFUL|) \ \mapsto \ p \in successful(K) \ \vee \ p \in active(K-1)$.

Proof Consider any state s at which $p \in active(K)$. It is sufficient to show that eventually $p \in done(K)$ since by procedure $Main$, at that time either process $p \in successful(K)$ or p will invoke $Toss$ with argument $K-1$. For this proof assume that $p \notin done(K)$ at state s. On account of requirement $(DET2)$ of $Detect$, p will eventually either find $count_p < K$ or exit upon executing a successful scan. In either case, $p \in done(K)$ holds eventually. □

Proposition 4 Starting from any state in which $p \in active(K) \ \wedge \ |\text{SUCCESSFUL}| = L$, within an expected finite number of rounds

$$p \in \text{SUCCESSFUL} \ \vee \ (p \in active(N-L-1) \ \wedge \ |\text{SUCCESSFUL}| > L \ ).$$

Proof Consider any state s at which $p \in active(K) \ \wedge \ |SUCCESSFUL| = L$ holds. By Lemma 6, $K \geq N-L$ at state s. Lemma 7 can be applied repeatedly to infer that eventually a state t is reached at which $p \in SUCCESSFUL \ \vee \ p \in active(N-L)$. At state t,

$(p \in SUCCESSFUL) \ \vee \ (p \in active(N-L) \ \wedge \ |SUCCESSFUL| > L) \ \vee$
$(p \in active(N-L) \ \wedge \ |SUCCESSFUL| = L)$.

We will show later that if the last disjunct holds then eventually a state u is reached at which

$$(p \in active(N-L) \wedge |SUCCESSFUL| > L) \vee (p \in active(N-L) \wedge |active(N-L)| = N-L) \ . \quad (3)$$

Combining the above two assertions, eventually,

$(p \in SUCCESSFUL) \ \vee \ (p \in active(N-L) \ \wedge \ |SUCCESSFUL| > L) \ \vee$
$(p \in active(N-L) \ \wedge \ |active(N-L)| = N-L)$.

If the first disjunct holds then there is nothing more to be proved. If the second disjunct holds, then we apply Lemma 7 to infer the desired condition. If the third disjunct holds then we apply Theorem 3 to infer that within an expected finite number of rounds, $|successful(N-L)| > 0$. Using Lemma 6 and the stability of membership in $active(N-L)$, it follows that within an expected finite number of rounds,

$(p \in active(N-L) \ \wedge \ |SUCCESSFUL| > L)$

holds. As in the proof of the second disjunct, we can now apply Lemma 7 to infer the desired condition.

For a proof of property (3), let q be any process $\notin SUCCESSFUL \ \cup active(N-L)$ at state t. (Property (3) follows immediately if there is no such process.) By applying Lemma 7 repeatedly to this process, eventually a state t_q is reached at which $q \in SUCCESSFUL \ \vee \ q \in active(N-L)$. Repeating this argument for every q, let u be the last of the states t_q. Either $|active(N-L)| = N-L$ or $|SUCCESSFUL| > L$ at this state. Since membership in $active(N-L)$ is stable, the proof of the property follows. □

Proposition 5 Any *useful* invocation R of $Rscan(n, K, i, success), n \leq K$, for which $(\forall p, q \in active(K) : rand_p[i] \neq rand_q[i])$ holds immediately prior to R, sets $success$ to true.

Proof We prove Proposition 5 by induction on n. First, consider the base case $n \leq 2$. In this case $Rscan(n, K, i, success)$ counts the number of ones in array $A_K[i].val$. Since R is a useful scan, there are no overlapping choose or initialize operations and $|active(K)| = K$ processes have chosen and set bits in $A_K[i].val$ to one prior to R. Furthermore, since $rand_p[i] \neq rand_q[i]$ for any pair p, q, the choices made by the processes are distinct. Therefore, $NumOnes(A_K[i].val, rowsize(K))$ returns K and consequently, R sets *success* to true.

For the induction step, we assume the hypothesis for $2, 3, \ldots n-1$ and prove it for n. Let R be any execution of $Rscan(n, K, i, success)$. R first calls $Rscan(n-1, K, i, success)$. Let R' denote this execution of $Rscan(n-1, K, i, success)$. Since R' is nested within R and R is useful, R' is also useful. Also, since all processes made distinct choices prior to R and $choosing\text{-}set(R)$ is empty, all processes also made distinct choices prior to R'. By the induction hypothesis, R' sets *success* to true. Therefore, the **then** branch of the conditional **if** statement in Figure 12 is chosen and $A_K[i].flag[n]$ is set to one. Consider any of the subsequent K iterations of the **while** loop. Note that a bit in $A_K[i].flag$ can be set to zero only by *Choose* and *Initialize* operations. Since R is a useful scan operation, all processes have completed their initializations and there are no overlapping *Choose* operations. Therefore, $A_K[i].flag[n]$ will remain set to one. Also, since there are no overlapping choose operations and all processes set distinct bits in $A_K[i].val$, $NumOnes(A_K[i].val, rowsize(K))$ will return K. Therefore, each iteration will maintain *success* as true. Consequently, R returns *success* as *true*. This completes the proof of the induction step and the proof of Proposition 5. □

Proposition 6 *For any execution R of $Rscan(n, K, i, success), 2 \leq n \leq K$, that sets success to true,* $|choosing\text{-}set(R)| \leq n-2 \Rightarrow [(\forall p, q : p, q \in active(K) \setminus choosing\text{-}set(R) : rand_p[i] \neq rand_q[i])$ at the state immediately prior to $R]$.

Proof We prove Proposition 6 by induction on n. First, consider the base case $n := 2$. Since R sets *success* to true, it follows from Figure 12 that R observed K distinct bits in row $A_K[i].val$. The antecedent implies that there are no choosing processes during the execution of R. Therefore, all processes must have set distinct bits in $A_K[i].val$ and the consequent follows.

For the induction step, we assume the hypothesis for $2, 3, \ldots n-1$ and prove it for n. Let R be any execution of $Rscan(n, K, i, success)$ by process p that sets *success* to true. Consider the iterations of the **while** loop in Figure 12. Since *success* is set to true, process p observes K ones in $A_K[i].val$ and finds $A_K[i].flag[n] = 1$ in each of the K iterations. If during any of these K iterations, all the ones that process p observed were written by distinct processes, then the consequent would follow as in the base case. Therefore we can assume that during each iteration of the **while** loop process p reads at least 2 ones written by the same process. For each iteration j, pick a process r_j with this property. This implies that process r_j performed a choose operation during R and from Figure 11, process r_j must have set bit $A_K[i].flag[n]$ to zero at some state u_j. Let s be the state at which process p sets bit $A_K[i].flag[n]$ to one and let t_j be the state at which process p observes the same bit to be one. It follows that $s \mapsto u_j \mapsto t_j$ for every j, $1 \leq j \leq K$. Since $A_K[i].flag[n]$ is 0 at state u_j and 1 at state t_j, there exists a process $q_j \neq p$ that sets the bit to one at some state v_j between the states u_j and t_j, i.e., $s \mapsto u_j \mapsto v_j \mapsto t_j$. By assumption, such a process q_j exists for each iteration of the **while** loop. Since there are K iterations and at most $K-1$ other processes, we can find two indices $a < b$ and a process q with $q = q_a = q_b$ (this explains the need for K iterations in Figure 12) and $s \mapsto v_a \mapsto v_b \mapsto t_b$, i.e., process q sets bit $A_K[i].flag[n]$ to one at state v_a and later at v_b during the execution of R. Observe from Figures 11 and 12 that process q sets a bit in $flag$ to one only during the execution of *Rscan*. In particular, bit $A_K[i].flag[n]$ is set to one only after q executes a $Rscan(n-1, K, i, success)$ operation R' that sets *success* to true. By the state sequence outlined earlier, process q must have executed R'

between the states v_a and v_b during the execution of R. Note that *choosing-set*$(R') \subseteq$ *choosing-set*(R) but *choosing-set*$(R') \neq$ *choosing-set*(R), since $q \in$ *choosing-set*$(R) \setminus$ *choosing-set*(R'). Consequently $|$*choosing-set*$(R')| \leq n-3$. Applying the induction hypothesis on R', we conclude that all processes in *active*$(K) \setminus$ *choosing-set*(R') made distinct choices prior to R'. Since *choosing-set*$(R') \subset$ *choosing-set*(R), all processes in *active*$(K) \setminus$ *choosing-set*(R) also made distinct choices prior to R'. Since these processes do not make any fresh choices during R and membership in *active*(K) is stable, processes in *active*(K) must have made distinct choices prior to R as well. This proves the proposition. □

Proposition 7 *Let x_N be the value of $x \geq N$ that minimizes $E_N(x) = x/p_N(x)$. Then*

1. $\lim_{N\to\infty} x_N \;=\; \frac{1}{2}N^2$,
2. $\lim_{N\to\infty} E_N(x_N)/x_N = e$ *where $e = 2.718..$ is the base of the natural logarithm.*

Proof Let $E_N(x) = x/p_N(x)$. Note that $p_N(x)$ can be written in the form

$$p_N(x) = \prod_{i=1}^{N-1} (1 - \frac{i}{x}) \ .$$

Let

$$s_N(x) = \sum_{i=1}^{N-1} \frac{i}{x-i} \ .$$

Then $p_N'(x) = \frac{1}{x} p_N(x) s_N(x)$. Setting $E_N'(x) = 0$, we find that the critical values of E_N are solution of

$$p_N(x) - x p_N'(x) = 0.$$

Since $x \geq N$, this forces

$$s_N(x) - 1 = 0. \tag{4}$$

The following lemma gives the behavior of $s_N(x)$ for $x = cN^2$ for some constant $c \geq 1/2$.

Lemma 8 *Given $\epsilon > 0$, there exists N_0 such that for $N \geq N_0$,* $\quad |s_N(cN^2) - \frac{1}{2c}| \ < \ \frac{\epsilon}{2}$.

Proof

$$\begin{aligned}
|s_N(cN^2) - \frac{1}{2c}| &= \left| \sum_{i=1}^{N-1} (\frac{i}{cN^2 - i} - \frac{1}{2c(N-1)}) \right| \\
&\leq \sum_{i=1}^{\lceil \frac{N-1}{2} \rceil} \left| \frac{i}{cN^2 - i} + \frac{N-i}{cN^2 - (N-i)} - \frac{1}{c(N-1)} \right| \\
&\leq \frac{N}{2}\, max_{1 \leq i \leq \frac{N}{2}} \left| \frac{i}{cN^2 - i} + \frac{N-i}{cn^2 - (N-i)} - \frac{1}{c(N-1)} \right| \\
&= max_{1 \leq i \leq \frac{N}{2}} \left| \frac{(c^2-c)N^4 + 2ciN^3 + ((1-2c)i - 2ci^2)N^2 + (2c-1)i^2 N}{2c(N-1)(cN^2 - i)(cN^2 - N + i)} \right|
\end{aligned} \tag{5}$$

Since the numerator in (5) is a quartic in N whereas the denominator is a quintic, it follows that for some positive constant C_0, we have

$$|s_N(cN^2) - \frac{1}{2c}| \ \leq \ \frac{C_0}{N} \ .$$

This proves the lemma. □

It follows that the asymptotic solution x_N to (4) is $x_N = \frac{1}{2}N^2$. Therefore part 1 of Proposition 7 holds. Part 2 of Proposition 7 is a consequence of the following lemma:

Lemma 9 $\lim_{N\to\infty} p_N(cN^2) = e^{-\frac{1}{2c}}$.

Proof This is a consequence of the inequalities

$$\int_1^N \log(1 - \frac{t}{cN^2})\, dt \;\leq\; \log(\, p_N(cN^2)\,) \;\leq\; \int_0^{N-1} \log(1 - \frac{t}{cN^2})\, dt$$

and the fact that each integral above converges to $-\frac{1}{2c}$. We omit the details. □

The second part of Proposition 7 now follows from Lemma 9 by taking $c = \frac{1}{2}$. □

Proposition 8

1. *Starting from any state in which* $|active(K)| = K$*, a state in which* $|successful(K)| > 0$ *is reached within*
$$O(K \cdot T_I(K) + \frac{\delta(K)}{p_K})$$
expected number of rounds.

2. *Starting from any state in which* $p \in active(K) \;\wedge\; |SUCCESSFUL| = L$*, a state in which*
$$p \in SUCCESSFUL \;\vee\; (p \in active(N-L-1) \;\wedge\; |SUCCESSFUL| > L\;)$$
is reached within
$$O(\frac{\delta(N-L)}{p_{N-L}} + (N-L)T_I(N-L) + \sum_{i=N-L}^{K}(1+\delta(i)))$$
expected number of rounds.

Proof
<u>Proof of part 1</u>
Part 1 of the proposition gives the expected number of rounds required for Theorem 3. We first consider the expected number of rounds required for Lemma 3, Lemma 4, and Proposition 3, which are used in the proof of Theorem 3.

(No. of rounds for Lemma 3):
We need to count the number of rounds required to guarantee that each process p makes a fresh choice on each row $A_K[i]$ of array A_K. From the code in Figure 3, this quantity is given by $\delta(K)$.

(No. of rounds for Lemma 4):
We count the number of rounds required to guarantee that the number of useful comparisons increases. From the proof of Lemma 4, this is the time required to complete a single iteration of the outer **repeat** loop of *Toss* in Figure 3, which is again $\delta(K)$.

(No. of rounds for Proposition 3):

We count the number of rounds between successive useful comparison of freshly picked values by each process belonging to $active(K)$. On account of the bounds for Lemma 3 and Lemma 4, the required number of rounds is $2 \cdot \delta(K) = O(\delta(K))$.

Now we can prove part 1 of Proposition 8. Within $K \cdot T_I(K)$ rounds processes complete the initialization of array A_K and within another $\delta(K)$ rounds *comp-init*(K) holds. Now, we apply Proposition 3 to get the sequence of useful scans $S_j, j \geq 0$. Then the expected number of additional rounds which guarantees that a process succeeds on array A_K is

$$O(\sum_{m \geq 1} p_K(1-p_K)^m \cdot m \cdot \delta(K)) \;=\; O(\frac{1-p_K}{p_K}\,\delta(K)) \;=\; O(\,\frac{\delta(K)}{p_K}\,)\ .$$

Therefore, the expected number of rounds required from a state at which $|active(K)| = K$ to a state at which $|successful(K)| > 0$ is

$$O(\delta(K) + K \cdot T_I(K) + \frac{\delta(K)}{p_K}) \;=\; O(K \cdot T_I(K) + \frac{\delta(K)}{p_K}).$$

Proof of part 2

Part 2 of the proposition gives the expected number of rounds required for Proposition 4. We first consider the expected number of rounds required for Lemma 7, which is used in the proof of Proposition 4.

(No. of rounds for Lemma 7):

Since $K > N - |SUCCESSFUL|$, process p will find $count_p < K$ within the time taken by successful processes to copy their choices on to B arrays and the time taken by process p to detect this fact. This implies that Lemma 7 holds within $1 + \delta(K)$ rounds.

Now we can prove part 2 of the proposition. Examining the proof of Proposition 4, the expected number of rounds to reach state t from state s is based on Lemma 7 and is given by

$$O(\sum_{i=N-L+1}^{K} (1 + \delta(i)))\ .$$

The expected number of rounds required for state u to hold from state t is also the above measure. Therefore, state u is reached from state s within the above number of rounds. If the first or the second disjunct of the proof holds, then by Lemma 7, the desired property holds within an additional $1 + \delta(N-L)$ rounds. If the last disjunct holds, then on account of Theorem 3, some process belongs to $successful(N-L)$ within

$$O(\frac{\delta(N-L)}{p_{N-L}} + (N-L)T_I(N-L))$$

number of rounds. Subsequently, by Lemma 7, the desired property holds within an additional $1 + \delta(N-L)$ rounds. Considering all three cases, the total expected number of rounds for the conclusion of Proposition 4 to hold is no more than

$$O(\frac{\delta(N-L)}{p_{N-L}} + (N-L)T_I(N-L) + \sum_{i=N-L}^{K} (1 + \delta(i)))\ .$$

□