Self-Stabilizing ℓ-Exclusion

Uri Abraham* Shlomi Dolev† Ted Herman‡ Irit Koll§

June 11, 2001

Abstract

Our work presents a self-stabilizing solution to the ℓ-exclusion problem. This problem is a well-known generalization of the mutual-exclusion problem in which up to ℓ, but never more than ℓ, processes are allowed simultaneously in their critical sections. Self-stabilization means that even when transient failures occur and some processes crash, the system finally resumes its regular and correct behavior. The model of communication assumed here is that of shared memory, in which processes use single-writer multiple-reader regular registers.

*Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. Email: `abraham@cs.bgu.ac.il`.

†Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. Partially supported by the Israeli ministry of science and arts grant #6756195. Email: `dolev@cs.bgu.ac.il`.

‡Department of Computer Science, University of Iowa. Email: `herman@cs.uiowa.edu`. This work is supported by NSF CAREER award CCR-9733541.

§Elta, Israel Ltd. and Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. Email: `iritk@is.elta.co.il`.

1 Introduction

The concept of self-stabilization was introduce by Dijkstra in [6], a work that proved to be the beginning of a flourishing subfield (see [9]). A self-stabilizing protocol is designed to cope with crashes and transient faults, and to resume normal operation after awhile. Normally, algorithms are supposed to start from some prescribed initial state (or set of states) in order to ensure their specifications, but a self-stabilizing protocol may start in any arbitrarily chosen initial state, and it will finally stabilize and recover its desirable properties. Clearly, a protocol designed to start in any state will also be able to cope with crashes and transient failures. To see that, just consider a state occurring after all transient failures have ended and when noncrashed processes operate normally.

Mutual exclusion is one of the fundamental problems in distributed computing [7], [18] and it is described in almost any textbook on operating systems or concurrency (e.g [22], [5]). It is defined for a system of n asynchronous processes which share a single resource: every process can access this resource, but only a single process can access it at a time.

Very few works consider self-stabilizing mutual-exclusion algorithms. One important exception is the seminal work [19]. Another self-stabilizing solution to the mutual-exclusion problem is mentioned in [21] and appears in [27]. In Dijkstra's self-stabilizing mutual exclusion protocol [6] the state variables are shared, and communication is constrained to a ring shape (See [8] for a more general topology).

The mutual exclusion problem was generalized to the ℓ-exclusion problem in [13]. In this problem the shared resource can be shared by at most ℓ processes at any time, for some specified $\ell \geq 1$. The program of every process has a piece of code called the *critical section* in which the process access this resource, and a solution of the ℓ-exclusion problem is a protocol that guarantees that at most ℓ processes have access to the critical section at any time.

The ℓ-exclusion problem can have several motivations. The motivation is evident if ℓ is the quantity of identical physical resources to be shared. For example, each process in a critical section may consume one unit of power, and only ℓ units of power are available. Variations on this theme are congestion control and bandwidth allocation; again, a process in the critical section has access to a fixed quota of bandwidth, and the total bandwidth should not exceed a limit given by ℓ. An ℓ-exclusion algorithm can also act as a "filter" to reduce concurrency for applications that specify limited concurrency.

The protocol of Dijkstra [6] is generalized in [12] to a self-stabilizing ℓ-exclusion algorithm designed for the ring shaped system. The algorithm in [12] does not satisfy all the requirements one would hope for in the common shared memory model where every process can communicate with every other process. In particular, all the processes in a ring must participate in executing the algorithm, even in the case they are not interested in entering the critical section. Roughly speaking, the processes pass a token in the ring and each process that holds the token may enter the critical section. Therefore, every process, whether it is interested in obtaining a resource or not, must participate in passing the privilege in the ring. The definition of the

ℓ-exclusion task allows up to $\ell - 1$ processes to crash. Hence, a self-stabilizing algorithm for the ℓ-exclusion task should tolerate crashes as well as transient faults, which the solution in [12] does not.

In this work we present for the first time a self-stabilizing ℓ-exclusion algorithm. (A preliminary version of this paper appears in [3].) The common shared memory model is the one considered in most works on mutual exclusion (e.g. [21, 23, 24, 26]), and this is also the model employed here in our solution. Our algorithm uses single-writer multi-reader *regular registers* which are of weaker type and therefore easier to implement than the basic atomic register. (These types of registers were defined by Lamport in [20].) The traditional approach using state machine and histories fits the self-stabilization paradigm, since self-stabilization is defined in terms of convergence of arbitrary initial states into legal states. On the other hand, states and histories are not very suitable for regular registers, since these are defined in terms of read/write events. This leads us to define a semantics that incorporates state machines, histories *and* system executions. As far as we know this is the first time (at least in the area of self-stabilization) that such an approach is taken.

The problem of self-stabilizing ℓ-exclusion which we solve here is mentioned as an interesting open problem in [4]. The solution given there to the ℓ-exclusion problem (namely the use of bounded timestamps) could also solve the self-stabilizing ℓ-exclusion problem—if self-stabilizing timestamps were known. However, at present, it seems that this is an open problem. Constructions of a bounded, concurrent timestamp system were provided in series of now classical works [15, 16, 11, 10]. Yet, none of these constructions is self stabilizing. Our present paper does not address the timestamp issue, instead we concentrate on the problem of ℓ-exclusion.

The remainder of the paper is organized as follows. In the next section we describe our model, define the notion of regular registers and formalize the assumptions and requirements for self-stabilizing ℓ-exclusion. The algorithm itself is presented in Section 8. It is called the *combined algorithm* since it is based on a couple of components that are carefully combined together.

2 The Self-Stabilizing ℓ-Exclusion Problem

2.1 System Executions

The model that we use here to describe and argue about distributed systems consists of two levels. The first is the standard state-transition approach, and the second is the system-execution approach. The combination of these two approaches is needed here since we have to argue about initial states and about moments (in order to tackle self-stabilization), and at the same time we want to argue about regular registers which are best described in terms of events and read/write executions.

Our ℓ-exclusion protocol is designed for processes that use regular registers, rather than serial (or atomic) registers. This is an advantage of our protocol, since it requires less from the system, and so we need a model in which regular registers can be defined.

We shall describe our model in levels of detail, and we begin with the first level. A distributed system consists of N processes $\{P_1, \ldots, P_N\}$. Each *process* executes a state machine. That is, the machine is the code executed by the process. A state of a process is defined by its program counter and the value of its *local variables.*

Formally, a state is a map from variables to values. A state of a system is a global state, namely a state that incorporates the program counters and local variables of all processes involved. In addition, a global initial state includes an initial value for each register. In the scope of self-stabilizing algorithms these initial values can be chosen arbitrarily.

Given a state, an instruction may be *enabled* or *disabled* at that state. Usually, several instructions may be enabled at any state. If S_1 and S_2 are (global) system states, and instruction I is enabled at S_1, then an execution of I can cause the transition from state S_1 to state S_2. Such a transition is called an *event*, or a *lower-level event* to mark its difference from higher-level events defined later.

The notation $S_1 \stackrel{e}{\to} S_2$ denotes the fact that S_1 and S_2 are states and event e caused the transition from S_1 to S_2.

A *history* of a system is a sequence $S_0, \pi_0, S_1, \pi_1 \ldots$ of system states S_i and events π_i where π_i is the transition from state S_i to S_{i+1}. A history represents the interleaving of events by different processes. The events $\pi_0, \pi_1, \ldots$ are instantaneous events. That is, each π_i is durationless. When later in this paper we refer to *moments* we shall mean lower-level events (which represent time instants). These lower-level events are linearly ordered in any history, and their "temporal" ordering is denoted with $\prec$. Thus $\pi_i \prec \pi_j$ iff $i < j$. We also use the notation $e \preceq f$ as a shorthand of $e \prec f$ or $e = f$.

This model of states and histories, with its variants and refinements, is used almost exclusively in today's literature on distributed systems, because of its simplicity and since it allows formal definitions and correctness proofs based on the assertional method. The encyclopedic textbook [22], for example, develops the i/o automata concepts and applies it to a great diversity of problems. In this paper we extend this state and histories model and allow events as well. As a result we can handle regular registers in the scope of self-stabilization.

Modeling atomic registers is simple: read/write events by the different processes are interleaved, and any read event returns the value of the most recent preceding write event. Regular registers, however, cannot be described so easily, and the read/write events are not represented as instantaneous occurrences but rather as temporal intervals. Lamport, who introduced these registers (in [20]), used system executions to define them. It is essential for the definition of regularity that read and write events are extended in time and are represented by temporal intervals rather than points. Thus two read/write events can intersect and overlap.

To model these registers with automata, we shall deal with start and conclude read events and with start and conclude write events, rather than with simple read/write events. Now, if

$S_0, \pi_0, \ldots$ is a history as before, and if for $i < j$ π_i is a start read and π_j its corresponding conclude read event (for example), then the interval $[\pi_i, \pi_j]$ of events in the history represents the execution of the extended read event. With every extended read event Y we associate a variable b_r that holds the set of possible values that this read may return. Suppose that $r = (\pi_i, \pi_j)$ is a read event. Then π_i is the start of r and the value of b_r is determined at π_i as follows:

- If there is no ongoing write (namely no start write without its corresponding conclude event precedes π_i) then $b_r = \{v\}$ where v is the value of the last compound write event on Y preceding π_i if such a write exists or the initial value of Y, otherwise.

- Otherwise, there is an ongoing write of value w on Y (namely there is a start write of value w without its corresponding conclude event that precedes π_i), then $b_r = \{v, w\}$ where v is the value of the last compound write event on Y preceding π_i, if such a write exists or the initial value of Y, otherwise.

Now every write event on Y adds its value to b_r until r is concluded. At conclusion the return value of the read r must be among the values in b_r.

The variables b_r are useful for us because self-stabilization is concerned with arbitrary states, and we must be able to tell from an arbitrary state what are the possible values that an ongoing read r may return. The corresponding value b_r gives us the answer. Since we consider arbitrary starting states and in particular a state in which a read event r has started already, we allow arbitrary values for such b_r's.

We note that every read r from a register of a crashed process returns the same value, namely the initial value of that register. This choice reflects the assumption that any write operation executed by a crashed process is virtually completed (even though the crash may occur in the middle of the write operation).

To argue in full generality we shall augment the history structure by making the following definition. A *compound event* is a set of lower-level events (in a given history). For example the set $\{st, cd\}$ of events, where st is a start read and cd the corresponding conclude read event, is a compound read event.

Recall that the lower-level events in any history are linearly ordered, and this "temporal" order is denoted with $\prec$. If A and B are compound events, then we define

$$A \prec B \text{ iff } \forall a \in A\ \forall b \in B\ (a \prec b) \tag{1}$$

Namely, compound event A precedes B if all events in the history that belong to A precede all events in the history that belong to B. For example, if $A = \{st_1, cd_1\}$ and $B = \{st_2, cd_2\}$ are compound read/write events, then $A \prec B$ iff $cd_1 \prec st_2$, namely A ends before B begins. We write $A \preceq B$ for $A \prec B \vee A = B$.

It turns out that the relation $\neg(B \prec A)$ is very important, as it corresponds intuitively to the notion that A can have an impact on B. To emphasize this fact we write $A \not\succ B$ for

$\neg(B \prec A)$. Now, symbol $A \not\succ B$ acquires an independent status and is read from left to right as "A weakly precedes B", or "A can influence B".

Let $\mathcal{H}$ be a history and $H = \{\pi_0, \pi_1, \ldots\}$ its set of events. Of course we do not take all subsets of H as compound events, but rather a subset E of relevant compound events is taken. As we saw above, pairs that describe extended read and write events are introduced into E. In addition, when some algorithm is investigated, the set of events pertaining to an execution of the protocol is collected and introduced into E. Such a compound event is also called a "higher level event" since it usually represents a higher level operation execution.

The resulting structure consisting of $\mathcal{H}$, E, and the temporal precedence relation $\prec$ defined above forms a *system execution*. The advantage of this two-level structure is that we can speak both about states and moments (namely lower-level events π_i are moments) and about compound events.

The following can easily be checked:

1. On the compound events of a given history, the precedence relation $\prec$ is an irreflexive partial ordering.

2. If A, B, C, and D are operations in the history such that
$$A \prec B,\ B \not\succ C, \text{ and } C \prec D,$$
then
$$A \prec D.$$
We will call this the Russell–Wiener property (as it was first investigated by Bertrand Russell and Norbert Wiener. See [25]).

3. Assuming that the compound events are pairwise disjoint and finite (sets of events) the following finiteness property holds.

    For any compound event X in E, there are only finitely many events Y in E for which $X \prec Y$ does not hold.

    We will call this the Lamport finiteness property (as it was first used by Lamport in [20]).

In fact, it is not necessary to assume that the events are pairwise disjoint, and it suffices to assume that each (lower-level) event belongs to only finitely many higher-level events in order to deduce the Lamport finiteness property.

Now we describe the semantics of regular registers. We use in this definition the function *Value*, which gives the value of a read/write compound event.

Definition 2.1 *Register Y is* regular *in a given history and its derived system execution if and only if for any compound read r of Y there is a compound write w on Y such that*

1. *Value*$(r) =$ *Value*(w), *and*

2. $w \not\rightarrow r$ *and there is no compound write* w' *onto* Y *such that* $w \prec w' \prec r$.

Equivalently, a register is regular in H *if there is a function,* ω*, defined on the (compound) read events of the register, and taking values that are (compound) write events of the register, such that the following holds:*

1. *Value*$(r) =$ *Value*$(\omega(r))$,

2. $\omega(r) \not\rightarrow r$ *and there is no compound write* w' *onto* Y *such that* $\omega(r) \prec w' \prec r$.

Such a function ω *is called a regular return function.*

2.2 The ℓ-exclusion problem

We make the following assumptions: a process in the critical section consumes a resource, and there are only $\ell < N$ resources in the system. Therefore the number of processes that execute the critical section is required to be bounded by ℓ.

Most of the time, a process executes a remainder section of its code which does not include any usage of the common resource; if and when it needs to use this resource it uses an ℓ-exclusion algorithm. This algorithm is a piece of code containing three sections: trying, critical, and exit. The trying section is the part that has to be executed before access to the resource is granted, the critical section is the access to this resource, and the exit section is an obligation which (among other things) tells the other processes that the resource is released. Thus the structure of the program of every process that executes the $\ell-exclusion$ algorithm is as follows:

```
repeat forever
    begin
        remainder section
        trying section
        critical section
        exit section
    end
```

A solution to the $\ell - exclusion$ problem is a program for the Trying and Exit sections that fulfills the $\ell - exclusion$ requirements.

The task of an ℓ-exclusion algorithm is to control the number of processes that are concurrently executing the critical section (this requirement is called the safety requirement) and let every process that executes the trying section to eventually enter the critical section (this is the liveness requirement). The exact formulation of these safety and liveness properties is rather subtle because of the possibility that processes crash, and so we discuss crashes first.

An important property of a solution for ℓ-exclusion is that it tolerates *crashes* of processes. A crashed process does not continue performing its code, and thus does not change the contents of its variables and registers. We assume that our execution starts following the last crash and therefore processes are either crashed or active. In fact our solution is suitable for any system in which there is a long enough period (subexecution) with no crashes (i.e., in such a period no process changes its status from being active to crashed).

¿From our point of view a process that crashes in its remainder section is indistinguishable from a process that is normally executing its remainder section forever — they both have the same values in their registers. Thus a process whose registers values show that it is in its remainder section is classified as being in its remainder section even if it is crashed and despite the value of its program counter which may point to another part of its algorithm. For the same reason, a normal process that stays forever in its critical section is indistinguishable from a process that crashes in its critical section (and its registers show that it is in that status). So a process that stays forever in its critical section is defined to be crashed.

The safety property of a solution for ℓ-exclusion problem is that there are never more than ℓ processes in their critical section, provided that the number of crashed processes does not exceed ℓ. That is, if there are $\ell + 1$, or more, crashed processes then no protocol can guarantee that they do not use up the shared resource that the protocol is supposed to control. If, however, the number of crashed processes is $n \leq \ell$, then we expect that the protocol will not let more than $\ell - n$ additional processes to be simultaneously in their critical section.

The liveness property for the usual mutual-exclusion problem (i.e. 1-exclusion problem) guarantees access to the critical section to any process, under the assumption that all critical section executions are terminating. Similarly, the ℓ-exclusion liveness property guarantees that processes will enter their critical section only if fewer than ℓ processes are crashed.

2.3 Self-Stabilizing ℓ-Exclusion

Self-stabilizing algorithms can be started in any arbitrary state. Therefore, they recover from occurrences of transient faults that leave the system in arbitrary states. One example for such an arbitrary state is a state in which every process is executing the critical section. Clearly, this state violates the ℓ-exclusion safety property. Therefore, stability requires only that *eventually* the system reaches a state after which no more than ℓ processes execute the critical section. A provision for this requirement is that the number of crashed processes in not exceeding ℓ.

In general, given a certain desirable property φ of histories, self stabilization of an algorithm with respect to φ means that the algorithm can be started in any arbitrary state and the resulting history will *finally* satisfy φ. That is, a suffix (a final segment) of the history will satisfy φ. Note that a suffix of a history is again a history.

For a self-stabilizing ℓ-exclusion algorithm, the requirement is that every computation has a suffix that fulfills the *safety* and *liveness* requirements of ℓ-exclusion. This gives the following properties.

Requirement 2.2 Safety: *If no more than ℓ processes are crashed then eventually at most ℓ processes are concurrently in the critical section at every system state.*

Requirement 2.3 Liveness (no starvation): *In an execution in which less than ℓ processes are crashed then each noncrashed process (that is in the exit section eventually executes the remainder section and each process) that enters the trying section eventually executes the critical section.*

To sum-up we have the following definition.

Definition 2.4 (Self-stabilizing ℓ-exclusion) *An ℓ-exclusion algorithm is self-stabilizing if every computation that starts with any arbitrary initial state satisfies the above safety and liveness requirements.*

3 The Main Techniques in a Nutshell

This section presents an overview of the main ideas and techniques of the algorithm. The algorithm that we are going to present in the sequel is rather complex and it has evolved gradually as simpler candidates were found inadequate. Several new ideas that were found to be necessary are combined in this work. As a result, we must admit, there is no simple description, yet a guide may help the reader. In the sequel we try to facilitate the understanding of the algorithm by presenting its different parts separately.

At the highest level, the algorithm can be seen as a combination of safety and no-starvation components. The idea of this paper, in a nutshell, is to introduce a mechanism to rotate identities among processes that compete for the critical section.

One main technique is what we call the *isolation algorithm.* This technique is to some extent a generalization of a technique used in [19] to guarantee fairness for mutual exclusion. The purpose of the isolation technique is to dynamically order processes in a way that ensures freedom from starvation. Shared variables for the isolation technique consist of an $N \times \ell$ matrix M of binary variables and a vector X of boolean variables. Variables in the i'th row of matrix M are written by P_i and read by all other processes. Each process P_i writes a boolean variable X_i to notify other processes that P_i is waiting to enter the critical section. Roughly speaking, the X variables eliminate from M the rows of processes not interested in the critical section. Each column in the remaining part of M dynamically defines a process identifier. The first column defines the identifier of the process considered to have the highest priority to enter the critical section. The second column defines the identifier of the process with second highest priority to enter the critical section, and so on.

How does a column of the remaining part of M define an identifier of a process? The following answer will later become a formal definition.

The *highest priority index* is defined according to the following rule: in case the value of the last variable in the column is identical to the value of the first variable in the column, then the index of the first row in the remaining part of M is chosen as the identifier of the process with the highest priority. Otherwise, let k be the minimal index of a row such that the value of the variable in this row is different from the value of the first variable in the column; k is the index of the highest priority process.

The second highest priority index is obtained from M after removing the first column and the row of the highest priority index from M. The second highest priority is defined by the same rule used to define the highest priority processes, but this time the rule is applied to the remaining matrix.

Due to the isolation technique a crashed process (a crashed process does not change its row in M) will eventually have high priority. Intuitively, the reason is that the rest of the processes will change the value of the variables in their rows.

To ensure that the algorithm fulfills the self-stabilizing ℓ-exclusion requirements we need to cope with several subtle issues. First we would like to ensure the safety property — namely, that eventually no more than ℓ processes execute the critical section. Note that the priorities implied by the isolation technique are not sufficient for this purpose; for instance, processes may assign their X variables *true*, which changes priorities of other processes. A *try* boolean variable is introduced to ensure that safety is achieved. Processes count the number of processes with $try_j = true$ as a safety test before entering the critical section. This counting must be done with care since the definition of ℓ-exclusion allows processes to repeatedly enter the critical section while other (say $\ell - 1$) slow processes are waiting for a resource. Slow processes that try to enter the critical section may unluckily read the *try* variables of the fast processes when each of them executes the critical section. Such slow processes may always count more than ℓ processes trying to enter the critical section, even though the fast processes are not executing the critical section simultaneously.

To ensure that eventually no more than ℓ processes are executing the critical section and at the same time ensure slow processes eventually enter the critical section, we use the *diagonal algorithm.* The idea of the diagonal algorithm is that a process P_i will count the processes that are trying to enter the critical section without considering processes that are too fast — "fast processes" give a clear indication that they have read the current *color* of P_i. We say that these fast processes *dominate* P_i.

One more technique is used in order to avoid deadlock. It is possible that, continuously, more than ℓ processes are trying to enter the critical section (although the set of trying processes changes over time). The processes with the lowest priority (according to the isolation technique) should set their *try* variables to *false*, thus letting other processes enter the critical section. Unfortunately, it is possible that the processes with lowest priority are crashed and will not set their *try* variables to *false*. To overcome this difficulty, each process P_i counts the processes having higher priority. Then P_i adds to this count the number of processes that do not dominate any process with higher priority. This method ensures that eventually a non-crashed process with the highest priority will not set its *try* to *false*; while other processes will calculate the number of processes that P_i will consider and set their *try* variable to *false* if necessary, letting P_i enter the critical section.

As we described above, the self stabilizing ℓ-exclusion protocol has two components, the *safe algorithm* and the *isolation algorithm*. These components are orthogonal and we shall describe them separately and then combine them into a single protocol. The properties proved for each component are then used to deduce the correctness of the *combined protocol*. The *safe protocol* itself is based on the *diagonal protocol* which is presented first in Section 4 and then used in the *safe algorithm* presented in Section 5.

A main ingredient is what we call the *isolation algorithm*. This, to some extent, generalizes a technique used in [19] to guarantee fairness for mutual exclusion. The *single column stabilization algorithm* is presented in Section 6 and then used in the *isolation algorithm* of Section 7. The final algorithm that uses both the diagonal and the isolation techniques is presented in Section 8. Section 9 discusses our result and poses several directions for further research.

4 The Diagonal Algorithm

There are N processes $P_1, \ldots, P_N$. Each process executes the following *diagonal protocol*

```
repeat forever
    begin
        remainder section
        diagonal algorithm
    end
```

Observe the distinction that we make here between the *diagonal protocol*, which is an infinite repeat loop, and the *diagonal algorithm* which is an ingredient in this loop. This distinction is made for convenience. The code for the *diagonal algorithm* appears in Figure 4. The remainder section may be either terminating or not.

Data Structures. An *index* is a process id index, that is an integer in $[1, \ldots, N]$. A *color* is an integer in $[1, \ldots, 2N]$. There are thus $2N$ colors. A *report-vector* by P_i is an array v of length N

$$v[1], \ldots, v[N]$$

such that $v[i]$ is a color number (also denoted $v.color$) and for $j \neq i$ $v[j]$ is a pair of color numbers

$$v[j].first,\ v[j].second.$$

The index i of v is written $id(v)$.

An "array of report-vectors" is an array $v_1, \ldots, v_N$ of length N such that each v_j is a report-vector by P_j.

If v and w are report-vectors by P_i and P_j respectively, then we say that v dominates w iff

$$w.color = v[j].\mathit{first} = v[j].second.$$

That is, both colors reported by $v[j]$ (where $j = id(w)$) agree with the color $w[j]$ of w.

A simple example illustrates these notions. Suppose that $N = 4$; so $color = \{1, \ldots, 8\}$. If two report vectors w and v by P_2 and P_4 are given, where

$$w = \langle (3,9), 5, (7,4), (1,1) \rangle$$

$$v = \langle (2,2), (5,5), (6,1), 1 \rangle$$

then $w.color = 5$, $v.color = 1$, $id(w) = 2$, $id(v) = 4$. Both w dominates v and v dominates w. Clearly, w would not dominate any report vector by P_1 because $w[1].\mathit{first} = 3 \neq 9 = w[1].second$.

Each process P_i writes in a regular register VEC_i which is read by all processes. The value of VEC_i is a report-vector by P_i. The initial value of VEC_i is arbitrarily determined by some initialization write.

The *diagonal algorithm* uses two local variables: v and *vector*. Variable v is an array of length N of report-vectors. The ith entry of v, v_i, is a report-vector by P_i (that is, $v_i[i]$ is a color and $v_i[j]$ for $j \neq i$ is a pair of colors). Variable *vector* holds report-vectors; it keeps its value from one invocation of the *diagonal algorithm* to the following (that is, *vector* is actually declared in the outer protocol).

The *diagonal algorithm* uses two procedures, *read-all-VEC*(v) which copies the VEC_j register into v_j for each j, and *report*(v) which uses parameter v to compute a report-vector. In detail, these procedures are described as follows (see Figures 1 and 2).

```
procedure read-all-VEC(v)

begin
1.  forall 1 ≤ j ≤ N concurrently do
2.  read (v_j := VEC_j);
end
```

Figure 1: Procedure *read-all-VEC* accepts a variable $v = v_1, \ldots, v_N$ where each v_j is a report-vector. The registers VEC_j are assumed to be regular.

The procedure *read-all-VEC*(v) fills v with values obtained from all registers VEC_j and in particular from VEC_i. In fact it makes no sense for P_i to read from its own register a value it may obtain from some local variable, but this style of writing is somewhat shorter and so we adopt it.

The order in which the registers are read (in *read-all-VEC*) is immaterial, and they may be read concurrently. They do not *have to* be read concurrently, but the absence of any prescribed order means that we do not need to assume such ordering. Just as we prefer to state our lemmas with the weakest possible assumptions, so we prefer to have concurrent statements, even in case the protocol is planned to be used by a sequential process.

We use the following notation to denote read/write instruction on registers: $\mathsf{read}(v := R)$ reads register R and assigns the value obtained to variable v, and $\mathsf{write}(R := v)$ writes value v on register R. If e is a read/write event (execution of some read/write instruction) and the value read or written is a report-vector, then $vector(e)$ denotes this value and $color(e)$ is the color entry of $vector(e)$.

The procedure *report* (see Figure 2) takes as parameter an array v of report-vectors $v_1, \ldots, v_N$ and it determines *vector* which is a report-vector by P_i. It first determines the "diagonal number" d which is a color that avoids the following colors: $v_i.color$ and all $2(N-1)$ colors in $v_j[i]$ for $j \neq i$. Then d is assigned to $vector.color$ (line 2). The other entries, namely $vector[j]$ for $j \neq i$, are determined to report on the last two colors obtained from P_j. Specifically, the first color of $vector[j]$ moves to become its second color, and the first color is updated to be v_j.color.

An execution of the *diagonal algorithm* (see Figure 4) consists of four parts: the set of initial reads (line **d1**), a calculation of *vector* (line **d2**), a write (line **d3**), and the set of last reads (in line **d4**). These last reads have no effect, and indeed the algorithm is not designed to perform anything particularly useful; rather it is an abstract investigation of an important component of the ℓ-exclusion algorithm.

procedure $report(v)$ (by P_i)
Parameter v is an array of report-vectors (entry v_i is by P_i).
Variable *vector* is of type report-vector by P_i.

begin
1. Let d be some color not in the following set of $\leq 2N-1$ colors:
   $\{v_i[i]\} \cup \bigcup_{j \neq i}\{v_j[i].first,\ v_j[i].second\}$.
2. $vector[i] := d$;
3. for every $j \neq i,\ 1 \leq j \leq N$,
   $vector[j].second := vector[j].first$;
   $vector[j].first := v_j[j]$;

end

Figure 2: Procedure $report(v)$, used by the diagonal algorithm.

Suppose that $\mathcal{S}$ is a system-execution for the *diagonal protocol*, in which each P_i executes the *diagonal algorithm* repeatedly a finite or infinite number of times. Since all read/write diagonal algorithm (by P_i)

```
begin
d1    read-all-VEC(v);
d2    report(v); (* variable vector is determined *)
d3    write(VEC_i := vector);
d4    read-all-VEC(v);
end
```

Figure 3: The diagonal algorithm by P_i. Variable *vector* is a report-vector by P_i.

events are terminating, if all remainder section executions by P_i are terminating, then there are infinitely many executions of the *diagonal algorithm*, but if there is a nonterminating remainder section, then it is not followed by any event and hence the number of executions of the *diagonal algorithm* by P_i is finite. This refers of course to a noncrashed process. A crashed process may fail while performing a read/write or any other operation. We shall deal with failures in the following subsection, and meanwhile we assume normal operations. Since the registers VEC_i are assumed to be regular, there exists a regular return function ω defined on the read events. If e is a read event in $\mathcal{S}$, then $\omega(e)$ is the write event affecting e. An execution of the *diagonal algorithm* (lines **d1**–**d4**) by P_i is called a "diagonal event" (it is a higher-level event). If g is a diagonal event by P_i then we write $i = id(g)$ and index i is called "the process of g". We denote by $vector(g)$ the report-vector value obtained in **d2** in g which is written on VEC_i. The color of $vector(g)$ is denoted $color(g)$. So $color(g) = vector(g)[id(g)]$.

The read events constituting the execution of line **d1** form an event which is denoted $\rho(g)$. For each $1 \leq j \leq N$, $\rho(g)$ contains a read of VEC_j event which is denoted $\rho_j(g)$.

The write event on VEC_i in the diagonal event g is denoted $\nu(g)$. Thus $\nu(g)$ is the execution of line **d3**.

The execution of **d4** in g is denoted $\sigma(g)$, and $\sigma_j(g)$ is the read of VEC_j in $\sigma(g)$.

So for every diagonal algorithm execution g, $\rho(g) \prec \nu(g) \prec \sigma(g)$. These sub-events are used in the formulations and proofs of the following lemma and corollary.

If $d_1,\ d2$ are diagonal events, then we say that

$$d_1 \text{ dominates } d_2$$

iff

$$vector(d_1) \text{ dominates } vector(d_2).$$

That is

$$color(d_2) = vector(d_1)[j].\mathit{first} = vector(d_1)[j].\mathit{second}$$

where $j = id(d_2)$.

Let d be a diagonal event and $j \neq k$ two indices. We say that

$$j \text{ dominates } k \;\text{ in } d$$

iff

$$\omega(\sigma_j(d)) \text{ dominates } \omega(\sigma_k(d)).$$

Where $\omega(\sigma_j(d))$ refers to the write event affecting $\sigma_j(d)$. Equivalently, j dominates k in d iff $vector(\sigma_j(d))$ dominates $vector(\sigma_k(d))$. (Recall that $vector(e)$ for any read event e is the vector value obtained by e.) In particular if $k = id(d)$ is the process of d, then the read $\sigma_k(d)$ returns the value of $\nu(d)$, and hence j dominates k in d iff $vector(\sigma_j(d))$ dominates $vector(d)$.

If g is a diagonal event by P_i and there is another diagonal event by P_i that follows g, then there is a first diagonal event by P_i following g. In this case we say that g has a "successor", and we denote it by g^+. Since its diagonal events are serially ordered, P_i has either a finite number of executions and then the final execution has no successor, or else it has an infinite number of executions in which case g^+ is always defined.

The following lemma is based on lemma 4.6 of [1] where the *diagonal algorithm* first appeared.

Lemma 4.1 *[The Diagonal Property] Let h and d be diagonal events by P_i and P_j respectively ($i \neq j$). Suppose that*

$$h \text{ dominates } d.$$

Let $r = \rho_j(h)$ be the first read event of VEC_j in h, and let $w = \nu(d)$ be the write event in d (onto VEC_j). Suppose that $r \prec w$. Then h^{++} exists and $\nu(h^{++}) \not\vdash s$, where $s = \rho_i(d)$ is the first read event of VEC_i in d.

Proof: We first explain the meaning of this lemma. If h and d are diagonal events by P_i and by P_j, and if $w = \omega(r)$ then we shall say that h reports on d (w and r are the write in d and read in h as in the lemma). If two successive diagonal events by P_i h_0 followed by h_1 both report on d, then clearly h_1 dominates d. It is possible however that h dominates d event if $h \prec d$ (or $r \prec w$). The lemma states that if this occurs, then h is quite old relatively to d. Namely h^{++} exists and $h^{++} \not\vdash d$.

Say $c = color(d)$. As h dominates d,

$$c = vector(h)[j].first = vector(h)[j].second. \tag{2}$$

Let $v \prec w$ be the write on VEC_j immediately preceding w (possibly v is the initial write on VEC_j). We first prove that $v \prec r$ is impossible. Assuming $v \prec r$, $v \prec r \prec w$ would imply that $v = \omega(r)$ and hence that

$$color(v) = vector(h)[j].first = c.$$

Yet the diagonal color $c = color(d)$ is supposed to avoid $color(v)$ (the previous color) and this contradicts $v \prec r$. Thus $r \not\succ v$ holds.

Let u_0 be the write on VEC_i immediately preceding $\nu(h)$. There is always such a write, for example, the initial write on VEC_i precedes $\nu(h)$. So

$$u_0 \prec r \not\succ v \prec s = \rho_i(d),$$

and hence $u_0 \prec s$ is deduced from the Russell–Wiener relation. Thus $u_0 \preceq \omega(s)$. We are going to rule out each of the following three possibilities for $\omega(s)$: $u_0 = \omega(s)$, $\nu(h) = \omega(s)$, $\nu(h^+) = \omega(s)$, and deduce the existence of h^{++} and the impossibility of $s \prec \nu(h^{++})$. Hence, $\nu(h^{++}) \not\succ s$ as claimed by the lemma.

We can deduce from (2) that

$$\begin{gathered} c \in vector(u_0)[j], \\ c \in vector(h)[j], \text{ and} \\ c \in vector(h^+)[j]. \end{gathered}$$

Since d avoids the two colors obtained from $\omega(s)$, $c = color(d)$ is not in $vector(\omega(s))[j]$. Thus $\omega(s)$ is not u_0, h, or h^+. Hence h^{++} exists and $\nu(h^{++}) \preceq \omega(s)$. So $\nu(h^{++}) \not\succ s$. ■

It is the following corollary rather than the lemma that will be used in the proof of the ℓ-exclusion property of our safe protocol.

Corollary 4.2 *Suppose that d is a diagonal event by P_j and $i \neq j$ is an index such that i dominates j holds in d. Let f be any diagonal event by P_i and suppose that $\rho_j(f) \prec \nu(d)$ (that is, the first read of VEC_j in f precedes the write onto VEC_j in d). Then f^+ exists and $\nu(f^+) \not\succ \sigma_i(d)$.*

Proof: Let $t = \sigma_i(d)$ be the second read of VEC_i in d. Since $\rho_j(f) \prec \nu(d)$ is assumed, $\rho_j(f) \prec t$. Consider the VEC_i write event $w = \omega(t)$ which affects t. There are three cases for w.

Case 1 $\nu(f) \prec w$. Then w is in f^+ or in some later diagonal event, and the conclusion of the lemma is obvious in this case. Namely $\nu(f^+) \not\succ t$ by regularity of VEC_i.

Case 2 $\nu(f) = w$. This case is impossible because it would imply that f dominates d and the previous lemma would give that f^{++} exists and $\nu(f^{++}) \not\succ \rho_i(d)$. Hence $f^+ \prec f^{++} \not\succ \rho_i(d) \prec \sigma_i(d)$ gives $f^+ \prec \sigma_i(d)$ which contradicts $\omega(t)$ being in f.

Case 3 $w \prec \nu(f)$ and then $w^+ = \nu(f)$ is the write on VEC_i following w. Since $vector(w)$ dominates $vector(d)$ (by assumption that i dominates j in d) it is not the case that w is the initial write on VEC_i, and hence w belongs to some diagonal event h. In fact $h^+ = f$. Now we use Lemma 4.1 to deduce that h^{++} exists and $\nu(h^{++}) \not\succ \rho_i(d)$. But $h^+ = f$, and hence $\nu(f^+) \not\succ \rho_i(d) \prec \sigma_i(d)$ which implies $\nu(f^+) \not\succ \sigma_i(d)$.

■

4.1 Self-stabilization of the diagonal algorithm

In analyzing self-stabilization we assume that the initial states of the processes are arbitrarily determined, with program counters, local variables, and registers having any possible values. Moreover, some of the processes are "crashed", which means that they never resume their activity, while others are active forever. Lemma 4.1 (and Corollary 4.2) need not remain true in this situation because we no longer make the assumption of appropriate starting states. For example, it is possible that h and d are diagonal events by P_i and P_j as in Lemma 4.1, and d evades its former color in variable $v_j[j]$ which is not its "real" former color! Namely, the initial state of P_j is such that the value of the local variable v_j does not correspond to the value of VEC_j. Such inconsistency, however, will be remedied as the system progresses. We will prove stabilization for Corollary 4.2.

Lemma 4.3 *Even in the presence of arbitrary initial states there is a moment τ such that Lemma 4.1 and Corollary 4.2 hold provided that the diagonal event d begins after τ.*

Proof: In any system-execution of the *diagonal protocol* we define three types of processes.

A0 A process is of type A0 if it is crashed (and hence inactive). A crashed process has no events, except its initial write that determines the value of its register.

A1 A process is of type A1 if it is active but finally in its remainder section (and stays there forever).

A2 A process is of type A2 if it is active and all executions of its remainder sections are terminating. The *diagonal algorithm* is executed an infinite number of times by a process of type A2.

We say that a diagonal event is of type $i = A0, A1, A2$ if its process is of type i (of course crashed processes have no events, but we refer to the assumed write of initial values of a crashed process as its diagonal event).

Since processes of type A0 never change the value of their registers, and processes of type A1 reach a stable value which is never changed again, there is a moment τ_0 such that if x is any diagonal event after τ_0 then x is of type A2 and x is not dominated by any event of type A0 or A1. Let τ be some moment after τ_0 such that any process of type A2 has executed at least two rounds of the *diagonal algorithm* in the interval τ_0 to τ. Now consider Lemma 4.1 with d assumed to be a diagonal event that begins after τ. If h dominates d as in the lemma, then h is of type A2 and we consider the following possibilities.

1. h begins before τ_0. Then the two events after h by P_j end before τ and show that $h^{++} \prec d$.

2. h begins after τ_0. Then the previous proof applies to give the desired conclusion.

Thus, the corollary holds provided that d is assumed to start after τ. ■

5 The safe algorithm

The *safe algorithm* described in this section is a self-stabilizing, ℓ-exclusion algorithm (that is still missing the required liveness properties). The *diagonal algorithm* presented earlier is an ingredient in this safe algorithm. Each process P_i executes the following *safe protocol*:

```
repeat forever
    begin
        remainder section
        Safe Algorithm
    end
```

Again, we make a distinction between the protocol and the algorithm which is described in Figure 4. In addition to the regular register VEC_i that carries report-vectors by P_i, process P_i possesses a boolean regular register TRY_i which can be read by all processes.

The *safe algorithm* begins by setting the local variable *try* to *false* and writing *false* in the regular register TRY_i. Then the repeat loop is executed. The body of this loop consists of lines **s3.1**–**s3.6**. An execution of the repeat-loop consists of one or more executions of its body; an infinite number of executions is also possible. Each execution of the repeat-loop's body begins with reading all *VEC* registers (procedure *read-all-VEC* is in Figure 1). Then procedure *report* determines the value of *vector* (*report* is defined in Figure 2). This is exactly how the *diagonal algorithm* begins, and we shall see below that the *diagonal algorithm* can be used as a component of the *safe algorithm*. Function *getTry*() returns a boolean value and determines *try*. This function is unspecified now, and one can think of it as a function that randomly determines the value of *try*. Later, when the *combined algorithm* is described, we shall specify this function and determine its properties. For the time being we make no assumption on *getTry* in order to prove the ℓ-exclusion property (other than the fact that it somehow returns a boolean value).

The value of *try* is written in TRY_i, and, in case *try* is *true* but was previously *false*, VEC_i is written. That is VEC_i is written only if *try* has changed its value from *false* to *true*. The condition on line **s4** is such that if *try* is *true* the loop terminates. If *try* is *false*, another execution of the loop's body is carried. A terminating execution of the repeat-loop ends with condition $try = true$; and we say that such an execution of the repeat-loop is *successful*.

Lines **s3**–**s7** form the "goto loop". The body of this loop contains the repeat-loop and the read instructions in **s5** which determine v and t. The code of *read-all-TRY*(t) appears in Figure 5. B is the set of indices j for which TRY_j was found *true* and the value from VEC_j does not dominate v_i (the value from VEC_i). Certainly $i \in B$ because TRY_i was written *true* and so t_i is true, and since (by definition) v_i does not dominate v_i. Condition $\mid B \mid \leq \ell$ ends the loop, whereas $\mid B \mid > \ell$ transfers control to the repeat loop. ($\mid B \mid$ is the cardinality of B,

```
safe algorithm (by P_i)

begin
s1    try := false;
s2    write (TRY_i := false);
s3    repeat
s3.1        read-all-VEC(v);
s3.2        report(v); (* variable vector is determined. *)
s3.3        old-try := try;
s3.4        try := getTry();
s3.5        write (TRY_i := try);
s3.6        if (try & ¬old-try) write (VEC_i := vector);
s4    until try;
s5    concurrently do { read-all-VEC(v), read-all-TRY(t)};
s6    Let B be the set of indices defined as follows
              j ∈ B iff t_j and ¬(v_j dominates v_i).
s7    if | B |> ℓ goto s3;
s8    CS;
s9    write (TRY_i := false);
end
```

Figure 4: Safe algorithm (by P_i). Procedure *read-all-VEC* is in Figure 1, and *read-all-TRY* is in Figure 5. Procedure *getTry*() is unspecified. Variables *try* and *old-try* are boolean, v and *vector* are as in procedure *report*(v) (Figure 2). Variable t is an array of length N of boolean values.

```
procedure read-all-TRY(t)

begin
1.  ∀ 1 ≤ j ≤ N concurrently do
2.  read(t_j := TRY_j)
end
```

Figure 5: Procedure *read-all-TRY*(t), t is an array of boolean values.

of course.) We say that the goto loop is "successful" if it ends with $\mid B \mid \leq \ell$, allowing access to the critical section CS. At exit, P_i writes *false* in TRY_i.

At this stage we may try to clarify the intuition behind the safe algorithm. The dominance relation is our substitute for timestamps, and condition v_j dominates v_i is an "indication" that P_j is after P_i. The lack of this indication, namely condition $\neg(v_j$ dominates $v_i)$, and condition t_j (which indicates the intention of P_j in writing *true* on TRY_j) are the two conditions that introduce j in B. Thus B contains the set of all indices of processes that seem to be willing to enter the critical section and show no indication that they are after P_i. Index i itself is in B. So if $\mid B \mid \leq \ell$ allows P_i to enter its critical section, there will not be more than ℓ processes there, since presumable all processes not in B will avoid entering their critical sections. The function *getTry*() that is specified later will allow processes to write *false* on their *TRY* register and thus to yield priority to other processes. This will turn out to be crucial for the liveness property.

A successful execution of the repeat loop must end in condition $try = true$, but executions of the repeat loop may begin either with $try = false$, for example the first iteration of the loop, or with $try = true$. For example if condition $\mid B \mid > \ell$ holds in **s7** and control returns to **s3**, then the repeat loop begins with $try = true$. An execution of the repeat loop by P_i, that writes in the VEC_i register, is said to be a "writing execution", and otherwise it is a "nonwriting execution".

Let r be any writing execution of the repeat loop by P_i. We shall define an execution of the *diagonal algorithm* contained in r and denote it $D(r)$. Consider the last execution of the body of the repeat loop in r, namely the last execution of lines **s3.1**–**s3.6**. Line **s3.1** executed there corresponds to **d1**, line **s3.2** corresponds to **d2**, line **s3.6** to **d3**, and the immediately following execution of *read-all-VEC*(v) in **s5** corresponds to **d4** (surely $try = true$ holds when **s4** is checked since r is a writing execution). We denote with $\rho(r)$ this execution of **s3.1**, with $\nu(r)$ the execution of **s3.6** (containing a write on VEC_i, since this is a writing loop), and with $\sigma(r)$ the execution of **s5**.

If we follow the series of values assumed by variable *try* in an execution of the repeat loop we find that it is of the following possible forms.

B1 $try = true$ throughout the execution. The loop contains a single cycle in this case, that is a single execution of the loop's body. This is a nonwriting execution.

B2 *try* may begin with *true* or *false*, but it is constantly determined *false* in executing line **s3.4**. Hence $try = false$ whenever **s4** is checked. This is a nonterminating, nonwriting execution.

B3 Variable *try* changes at some stage from *false* to *true*. There are two possible sub-forms in this case:

B3a $try = false$ first, and at some stage it changes to $try = true$. This is a writing execution.

B3b $try = true$, then changes to *false* (in **s3.4**), and after a certain time it ends with *true* again. This is a writing execution.

An execution of the goto loop is called *normal* if it is preceded by an execution of line **s1** that determines $try = false$. Any normal execution of the goto loop thus begins with variable *try* set to *false*. We consider abnormal executions of the goto loop as well, since for self-stabilization we must consider cases in which the execution begins after **s1** and with variable *try* set to *true*. Thus any normal and successful execution of the goto loop contains a writing execution of the repeat-loop, and hence a last writing execution. It is possible that this last writing execution is followed by nonwriting executions (which must be of form B1: namely the value of *try* is always *true* there).

Let p be any normal and successful execution of the goto loop by P_i. Look at the last writing execution r of the repeat loop in p (this is the execution containing the last write in VEC_i) and define $D(p) = D(r)$. We write $\rho(p)$ for $\rho(r)$, $\nu(p)$ for $\nu(r)$, and $\sigma(p)$ for $\sigma(r)$. The execution of **s3.5**, $\mathsf{write}(TRY_i := try)$, between $\rho(p)$ and $\nu(p)$ is denoted $\tau(p)$. Note that, $\rho(p) \prec \tau(p) \prec \nu(p) \prec \sigma(p)$. Clearly $\tau(p)$ writes *true* on TRY_i, and there is no execution of **s3.5** afterwards in p that writes *false* (since we are considering the last writing repeat loop).

An execution of the algorithm that begins from line **s1** is said to be *normal.* Since processes may start in any state, abnormal executions may exist.

Let $\mathcal{S}$ be a system execution of the *safe protocol.* We define four types of processes.

C0 A process is of type C0 if it is crashed and its *TRY* register is *true.*

C1 A process is of type C1 if either it is active and finally in its remainder section or else it is crashed and its *TRY* register is *false.* Active processes in their remainder section set their *TRY* register to *false*, so in both cases the register of a process of type C1 is finally *false.*

C2 A process is of type C2 if it is active and finally locked in the goto loop.

C3 A process is of type C3 otherwise. Processes of type C3 are active and have only successful goto loops. Processes of type C3 always terminate their critical sections (or else they are crashed), and thus have an infinite number of critical section executions.

If c is any critical-section event by P_i that belongs to a normal execution of the algorithm, then c is preceded by a successful execution p of the goto-loop. We write $D(c)$ for $D(p)$, namely for the diagonal event preceding c. We write $\rho(c)$, $\nu(c)$, and $\sigma(c)$ for $\rho(p)$, $\nu(p)$, and $\sigma(p)$ respectively. Similarly $\tau(c) = \tau(p)$ is the last write of *true* in TRY_i in p.

Now we are ready to prove the self-stabilizing ℓ-exclusion property in any system execution $\mathcal{S}$.

Lemma 5.1 *The safe algorithm satisfies the Safety Requirement (2.2). Namely that if no more than ℓ processes are crashed then eventually at most ℓ processes are concurrently in the critical section at every system state.*

Proof: Let $\mathcal{S}$ be a system execution of the safe algorithm. Consider the crashed processes and let C be the set of indices of processes of type C0. Assume $\mid C \mid = q \leq \ell$. Let τ be a moment as in Lemma 4.3. By choosing τ far enough, we may assume that if c is any critical section event beginning after τ, then c belongs to a normal execution of the algorithm. Observe that τ has this property that if d is a diagonal event by P_j beginning after τ and i dominates j holds in d, then P_i is not crashed.

Given a set of $\geq \ell + 1 - q$ critical section events in $\mathcal{S}$ occurring after τ, we must find two that are comparable (one precedes the other). We may assume that these CS events belong to different processes (since each process is serial). Let K be a set of indices of these processes. So $\mid K \mid = \ell + 1 - q$ and suppose that c_j for $j \in K$ is a CS event by P_j occurring after τ. We shall find $i \neq j$ in K with $c_i \prec c_j$.

Consider $d_j = \tau(c_j)$, the last write of *true* in TRY_j before c_j, and let $i_0 \in K$ be such that $\mathsf{conclude}(d_{i_0})$ is maximal among all $\mathsf{conclude}(d_j)$, $j \in K$. Now the set of indices $B = B_{c_{i_0}}$, as determined by P_{i_0} in executing **s6** has finally size $\leq \ell$ since the goto loop is successful. Observe first that all indices of type C0 are in B. That is $C \subseteq B$. This is so because if j is of type C0, then P_{i_0} finds in defining B that t_j holds and v_j does not dominate v_{i_0} (since P_j is crashed). Since all events in K are alive, $K \subseteq B$ is impossible, because it leads to $\mid B \mid \geq \mid C \mid + \mid K \mid = q + \ell + 1 - q > \ell$. Thus $K \setminus B$ is nonempty, and we may pick some index $j \in K \setminus B$. We shall prove that $c_j \prec c_{i_0}$.

Let r be the read of TRY_j in $\sigma(c_{i_0})$, and let $v_{i_0} = \nu(c_{i_0})$ be the last write in VEC_{i_0} before c_{i_0}. So $d_{i_0} \prec v_{i_0} \prec r \prec c_{i_0}$. As $\mathsf{conclude}(d_j) \prec \mathsf{conclude}(d_{i_0})$, $d_j \prec r$, and hence $d_j \preceq \omega(r)$ where $\omega(r)$ is the write in TRY_j affecting r. If r returns *false*, then $d_j \preceq \omega(r)$ would clearly imply $c_j \prec c_{i_0}$ (as d_j is a write of *true*, and $\omega(r)$ is a write of *false*, $\omega(r)$ is after the CS event c_j, and we would have $c_j \prec \omega(r) \not\succ r \prec c_{i_0}$ and hence $c_j \prec c_{i_0}$). So r obtains *true* and hence $t_j = true$ holds in the execution of **s6**. Yet $j \notin B$ and hence j dominates i_0 in the diagonal event $D(c_{i_0})$ of c_{i_0}.

Let s be the read of VEC_j done by P_{i_0} in $\sigma(c_{i_0})$. Then $v_{i_0} \prec s \prec c_{i_0}$. Since d_j ends before d_{i_0} does, $d_j \prec s$ can be concluded.

Let $v_j = \nu(c_j)$ be the write in VEC_j, and let t be the read of VEC_{i_0} done by P_j before d_j in $\rho(c_j)$. Then $t \prec d_j$, and as $d_j \prec \mathsf{end}(d_{i_0}) \prec v_{i_0}$, $t \prec v_{i_0}$. Thus, corollary 4.2 can be used to conclude that v_j^+ exists and

$$v_j^+ \not\succ s.$$

Yet, $c_j \prec v_j^+$ and $s \prec c_{i_0}$, implying $c_j \prec c_{i_0}$. ■

6 The Single Column Stabilization Algorithm

This section is a prelude to the ℓ-dimensional algorithm of the following section (the *isolation algorithm*). The single dimension algorithm presented here is simpler and is given with more details. Since the data type used here is a single column rather than a matrix, this simpler algorithm is called *single column* and is denoted SC.

Data types. An ord-value is one of $\{0, 1, *\}$. Here $*$ is the "inactive" or "undefined" value. An ord-column is an array

$$J = \begin{pmatrix} J_1 \\ \vdots \\ J_N \end{pmatrix}$$

of length $N \geq 2$ of ord entries. The column is "nonempty" if at least one entry is defined (i.e., not $*$). An *index* is one of $\{1, \ldots, N\}$. So the first index in the column is 1 rather than 0.

Let J be a nonempty ord-column. The set of indices $1 \leq k \leq N$ for which $J_k \neq *$ is denoted $\mathit{Active}(J)$. Thus J is nonempty iff $\mathit{Active}(J) \neq \emptyset$. We define for a nonempty ord-column J:

$$\mathit{first}(J) = \min(\mathit{Active}(J))$$

$$\mathit{last}(J) = \max(\mathit{Active}(J)).$$

So $\mathit{first}(J) = k$ iff k is the least index for which $J_k \neq *$. And $\mathit{last}(J) = k$ iff k is the highest index for which $J_k \neq *$.

If $\mid \mathit{Active}(J) \mid \geq 1$ and $1 \leq i \leq N$, then the index $\mathit{previous}(i, J) \in \mathit{Active}(J)$ is defined as follows. If $i \leq \mathit{first}(J)$, then $\mathit{previous}(i, J) = \mathit{last}(J)$, but if $i > \mathit{first}(J)$, then $\mathit{previous}(i, J)$ is the maximal active index strictly below i.

function Choice(J)

1. Let $m = \mathit{first}(J),\ n = \mathit{last}(J)$;
2. if $J_m = J_n$ return m;
3. else let k be the least active index in J such that $J_k \neq J_m$
4. return k

Figure 6: Function $\mathit{Choice}(J)$, with parameter J that is an ord-column.

It makes some of our definitions clearer to arrange the index set $\{1, \ldots, N\}$ clockwise in a ring, with 1 following N. To find $\mathit{previous}(i, J)$ go from i counter clockwise (one or more steps) until a J index is reached. So, for example, if $\mathit{active}(J) = \{k\}$ is a singleton, then clearly $\mathit{previous}(i, J) = k$ for every i.

Figure 6 defines the choice function, $\mathit{Choice}(J)$, which receives a nonempty column J and returns an index in $\mathit{Active}(J)$, called the chosen index. In case $\mid \mathit{active}(J) \mid = 1$, $\mathit{choice}(J)$ returns that single active index. If $\mathit{first}(J) = m < n = \mathit{last}(J)$, then the values $J_m,\ J_n$ are checked. If they are the same (0 or 1) then the first active index m is returned, but if they differ then we know that there is some active index $k > m$ such that $J_k \neq J_m$ (namely $k = n$) and the least such k is returned. See Figure 7 for an example with explicit values.

$$J = \begin{pmatrix} * \\ * \\ 1 \\ 1 \\ * \\ 0 \\ * \\ * \\ * \end{pmatrix}$$

Figure 7: An example of an ord-column. Here $N = 9$. $active(J) = \{3, 4, 6\}$, $first(J) = 3$, $last(J) = 6$. $previous(k, J) = 6$ for $k = 1, 2, 3$. $previous(4, J) = 3$, $previous(5, J) = 4$, $previous(6, J) = 4$, and $previous(k, J) = 6$ if $k = 7, 8, 9$. $choice(J) = 6$.

We now define the notions of extensions and conservative extensions. If v and w are two nonempty ord-columns we say that w extends v if $active(v) \subseteq active(w)$ and $w[i] = v[i]$ for every $i \in active(v)$.

We say that ord-column w is a *conservative* extension of nonempty ord-column v if w extends v and for every index $k \in active(w) \setminus active(v)$ if $first(v) < k$ then

$$w[k] = v[previous(k, v)], \text{ and}$$

if $k < first(v)$ then

$$w[k] = 1 - v[last(v)].$$

The following observations follow immediately from the definition and are marked down for future reference.

Lemma 6.1 *(1) If v is a nonempty ord-column then there is a uniquely defined ord-column w with $active(w) = \{1, \ldots, N\}$ such that w is a conservative extension of v.*

(2) If w is a conservative extension of v, and u is an ord-column such that u extends v and w extends u, then u is a conservative extension of v and w is a conservative extension of u.

(3) A conservative extension of a conservative extension of v is a conservative extension of v. (Thus if w_1 and w_2 are two conservative extensions of v and $k \in active(w_1) \cap active(w_2)$, then $w_1[k] = w_2[k]$.)

(4) If w and v are nonempty ord-columns and w is a conservative extension of v, then

$$choice(w) = choice(v).$$

Proof: We prove the fourth item by following the definition of $choice(v)$. Let $m = first(v)$ and $n = last(v)$. Two facts are needed here:

1. For every $n' \geq n$ in $active(w)$, $w[n'] = v[n]$. In particular, if $n_0 = last(w)$ then $w[n_0] = v[n]$.
2. If $m_0 = first(w)$ then either $m_0 = m$, or $m_0 < m$ and for every $m_0 \leq m' < m$ in $active(w)$ $w[m'] = 1 - v[n]$.

According to the way $choice(v)$ is defined, there are two cases.

Case 1: $v[m] = v[n]$ and so $m = choice(v)$. There are two possibilities for w. If $m_0 = m$, then $w[m_0] = w[n_0]$ follows and so $m_0 = choice(w)$ as well. If $m_0 < m$ then the second fact above implies that m is the first index for which $w(m) \neq w(m_0)$ and hence $choice(w) = m$.

Case 2: $v[m] \neq v[n]$. Then $k = choice(v)$ is the first active index such that $v[k] \neq v[m]$. It is not difficult to prove that k is the first index in $active(w)$ as well such that $w[k] \neq w[m_0]$. Therefore, $k = active(w)$. ■

Definition 6.2 *Let J be a nonempty ord-column and k an index. The function $change\_order(J,k)$ returns the following $\{0,1\}$ value. Let $p = previous(k,J)$. If $p < k$, then $change\_order(J,k) = J_p$, but if $k \leq p$, then $change\_order(J,k) = 1 - J_p$.*

Equivalently, we can say that if $k \leq first(J)$, then $change\_order(J,k)$ reverses the value of $last(J)$, but if $k > first(J)$, then $change\_order(J,k)$ is obtained by copying the value of $previous(k,J)$.

For example, if

$$J = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ * \end{pmatrix}$$

then
$change\_order(J,1) = 1$, because $previous(1,J) = 4$ and $1 - J_4 = 1$.
$change\_order(J,2) = 0$, because $previous(2,J) = 1$ and $J_1 = 0$.
$change\_order(J,3) = 1$, because $previous(3,J) = 2$ and $J_2 = 1$.
$change\_order(J,4) = 1$, because $previous(4,J) = 3$ and $J_3 = 1$.
$change\_order(J,5) = 0$, because $previous(5,J) = 4$ and $J_4 = 0$.

6.1 The Single Column Algorithm

The *single column algorithm* (SC algorithm) is described in Figure 8. There are N processes $P_1, \ldots, P_N$ as before. Each process P_i repeats loops of the SC algorithm followed by some external activity which is called Remainder. The Remainder may be nonterminating. So P_i executes the following protocol.

```
repeat forever
begin
SC algorithm
Remainder
end
```

The registers used by P_i are

1. X_i which is boolean and regular, written by P_i and read by all processes.

2. ORD_i is regular, written by P_i and read by all processes. The values that ORD_i can assume are $\{0, 1\}$.

single column algorithm (by P_i)

begin
i1 $X_i := true$;
i2 $fill(J)$;
i3 if $return()$ goto **i1**;
i4 $X_i := false$;
i5 $v := change\_order(J, i)$;
i6 write($ORD_i := v$);
end

Figure 8: The SC algorithm for P_i. J is an ord-column, $fill(J)$ is in Figure 9, $change\_order(J, i)$ is in Definition 6.2, and $return()$ is an unspecified boolean function.

In executing the SC algorithm, P_i first sets X_i to be *true* and then fills column J. Procedure $fill(J)$ is in Figure 9. the execution of the $fill(J)$ procedure starts with a read operation of each X_j ; if X_j is *false* J_j is assigned by $*$, but otherwise the value of ORD_j (0 or 1) is read and assigned to J_j.

Then an unspecified boolean function $return()$ is applied to determine whether to return to **i1** or to continue. Lines **i1**–**i3** form the "goto" loop. At exit from this loop X_i is reset to *false*

```
procedure fill(J)

forall j concurrently do
    begin
        read(x_j := X_j);
        if x_j read(J_j := ORD_j) else J_j := *
    end
```

Figure 9: The *fill* procedure. J is an ord-column.

and then the value of $change\_order(J, i)$ is written in ORD_i. Function *change_order* is defined in Definition 6.2.

Let $\mathcal{S}$ be a system execution of the SC protocol. The initial state of each process and the initial value of each register are determined arbitrarily. We define two types of processes.

D0 A process is said to be *frozen* if it finally never changes the values of its registers. (That is, from some moment forth a frozen process either does not write in its registers or else it writes the same value over and again.) A frozen process is said to be frozen with $X = true$ if the final value of its X register is *true*. There are three sub-types of frozen processes.

- **D0a** Process P_i is crashed; X_i and ORD_i may have any (fixed) values as determined by some initializing event.
- **D0b** Process P_i is active but (finally) in a nonterminating remainder section. In this case X_i is finally *false* and ORD_i has some fixed value in some final section of $\mathcal{S}$. (Though P_i is in its remainder section it is active and it can fulfill its obligation to keep X_i *false*.)
- **D0c** The third sub-type of frozen processes are active and (finally) locked in a nonterminating execution of the goto loop. A process of sub-type D0c sets $X_i = true$ (finally) and its ORD register never changes in this loop.

D1 A process is not frozen if and only if it has only terminating goto loops and terminating Remainders. An unfrozen process thus contains an infinite number of SC executions in $\mathcal{S}$.

Processes of type D0 are called *frozen* since they do not change the values of their registers in some final segment of $\mathcal{S}$. The "frozen column" F is defined (in $\mathcal{S}$) as follows. For any index i, F_i is defined (i.e. not $*$) iff P_i is frozen with $X_i = true$, and the value of F_i in this case is the final value of ORD_i.

Thus the frozen column is such that if Z is the set of frozen processes P_i with $X_i = \mathit{true}$, then $Z = \mathit{active}(F)$. If $Z = \emptyset$ (i.e. there are no frozen processes) then the frozen column is void and the following theorem is trivially true.

Theorem 6.3 *Let F be the frozen column of $\mathcal{S}$. There is a moment τ such that any execution of fill(J) beginning after τ, determines a value of J that is a conservative extension of F.*

Proof: Let Z be the set of indices of frozen processes with $X = \mathit{true}$, and F be the corresponding frozen column. If F is empty, then the theorem is trivially true, since any column is a conservative extension of the empty column. So assume that F is nonempty.

Let τ_0 be a moment occurring after each $i \in Z$ is frozen. Thus, any read of register X_i for $i \in Z$ returns *true*, and any read of ORD_i returns F_i if they occur after τ_0.

We first prove that any execution of *fill*(J) after τ_0 determines that J extends F. Indeed, for any index $i \in \mathit{active}(F) = Z$, any read of X_i after τ_0 returns *true*, and any read of ORD_i returns the frozen value F_i. Hence, $J_i = F_i$ for $i \in Z$, and so J extends F.
Now we prove the existence of τ as in the theorem.

Let G be the fully defined (no $*$ entries) ord-column that is a conservative extension of F. The existence of G was claimed in Lemma 6.1(1).

If $j \notin Z$ then P_j is of type D3 and it executes the SC algorithm an infinite number of times. We shall prove that for every index $j \notin Z$ there is a moment $\tau(j)$ such that the following holds for every execution e of the SC algorithm (by P_j) occurring after $\tau(j)$: The value $v = \mathit{change\_order}(J, j)$ (determined in executing **i5**) in e is such that $v = G_j$. If we prove this, then define τ to be a moment later than τ_0 so that each P_j for $j \notin Z$ has executed a turn of the SC algorithm after $\tau(j)$. τ is as required now, since any execution of *fill*(J) obtains a value for J that extends F and at the same time is extended by G. Hence this value is a conservative extension of F (Lemma 6.1(2)). In details the argument goes as follows. Let E be an execution of *fill*(J) done after τ. For any index $j \in Z$, E obtains for J_i the frozen value F_j. For any index $j \notin Z$, J_i may well be $*$. Otherwise, in case E obtains *true* in reading X_j, then it is the value of ORD_j written by some execution of the SC algorithm that E obtains, and execution that is done after $\tau(j)$. Hence E determines J_j to be G_j.

So assume that $j \notin Z$ and we shall define $\tau(j)$ as required. The first sub-case that we consider is that $j > \mathit{first}(F)$. The definition is by induction on the distance from j to $\mathit{previous}(j, F)$. The base case is that $j \notin Z$ but $j - 1 \in Z$. Since $j \notin Z$, P_j is unfrozen, and we consider an execution e of the SC algorithm by P_j that starts after τ_0. We know that the value of J obtained in executing *fill*(J) in e extends F, and hence $\mathit{change\_order}(J, j)$ returns F_{j-1} which is G_j. So $\tau(j) = \tau_0$ in this case.

Next consider any $j \notin Z$ such that $j - 1 \notin Z$ but $j - 2 \in Z$, and let $\tau(j)$ be any moment occurring after a complete execution of the SC algorithm by P_{j-1} after $\tau(j-1)$. If e is an execution of the SC algorithm by P_j after $\tau(j)$, then *fill*(J) may find $X_{j-1} = \mathit{true}$ and it may also find $X_{j-1} = false$. However, in both cases $change\_order(J, j)$ (executed in e) returns $G_j = F_{j-2}$.

The proof continues similarly by induction, until, for every $j > first(F)$, a moment $\tau(j)$ is defined as required: any execution of the SC algorithm after $\tau(j)$ by P_j determines a value $v = change\_order(J, j)$ (in executing **i5**) such that $v = G_j$.

Now consider the sub-case where $j < first(F)$. The argument begins with process P_1 and continues inductively. Assume that $1 < first(F)$ and let $\tau(1)$ be a moment occurring after each process P_j for $j \notin Z$ and such that $j > first(F)$ has executed a round of its SC algorithm after $\tau(j)$. Consider an execution e of the SC algorithm by P_1 after $\tau(1)$. Let $n = last(F)$. For any index j such that $n \leq j \leq N$, if e contains a read of ORD_j then it obtains the value G_j. Yet all of these values are equal to F_n. Hence, $changeOrder(J, 1)$ executed in e returns the value $1 - F_n = G_1$. Now for indices $1 < j < first(F)$ the proof continues as above and the conclusion of our theorem is clear. ■

7 The isolation algorithm

The *isolation algorithm* extends the *single column algorithm* of the previous section to the multiple column (matrix) case.

Data types. An ord-row (or vector) is an array of length ℓ of values from $\{0, 1, *\}$. As before $*$ is the "inactive" or "undefined" value. An ord-row is "inactive" iff all its entries are $*$. A $\{0, 1\}$ row is said to be "active".

Ord-columns were defined in the previous section as arrays of length N of entries from $\{0, 1, *\}$.

An ord-matrix is an $N \times \ell$ array of entries from $\{0, 1, *\}$ such that each row is either inactive or active. Namely each row is either the $*$-row, or else contains no $*$ entries at all.

The matrix is said to be *nonempty* if it has at least one active row. If M is an ord-matrix, then $M(i, j)$ denotes the ith row, jth column entry of M (for $1 \leq i \leq N$, $1 \leq j \leq \ell$). The ith row is denoted M_i

$$M_i = M_i[1], \ldots, M_i[\ell]$$

where $M_i[k] = M(i, k)$. The jth column is denoted M^j.

If M is any ord-matrix and row M_m is active, then $inactivate(M, m)$ is the matrix M' obtained by deactivating row m. That is

$$M'_k = M_k, \text{ for } k \neq m.$$

$$M'_m[j] = *, \text{ for all } 1 \leq j \leq \ell.$$

For a nonempty ord-column J we have defined (in Figure 6) $choice(J)$, and now we shall define $choice(M)$ for ord-matrices M.

Definition 7.1 *Let M be a (nonempty) ord-matrix and let $\ell^* \geq 1$ be the number of its active rows. (Since M contains at least one active row, all its columns are nonempty). Define $choice(M)$ as the sequence of length $\ell' = \min\{\ell, \ell^*\}$ of pairs of matrices and pairwise distinct indices*

$$choice(M) = \langle (M(i), m_i) \mid 1 \leq i \leq \ell' \rangle$$

defined inductively as follows.

Let $M(1) = M$ be the given nonempty matrix. Let J^1 be the first column of $M(1)$. Define

$$m_1 = choice(J^1).$$

If $\ell' = 1$ then the defining process stops. Otherwise, let $M(2) = inactivate(M(1), m_1)$. Then $M(2)$ is nonempty (as $\ell^* > 1$). Let J^2 be the second column of $M(2)$ and define

$$m_2 = choice(J^2).$$

Now, for the general inductive definition, suppose that indices $m_1, \ldots, m_i$ and nonempty matrices $M(1), \ldots, M(i)$ have been defined, and $M(i)$ has $\ell' - i + 1$ active rows (at least). If $i = \ell'$ the procedure ends. Assuming that $i < \ell'$, $M(i)$ has $\ell' - i + 1 \geq 2$ active rows. Define

$$M(i+1) = inactivate(M(i), m_i)$$

Then $M(i+1)$ is nonempty, and it has $\ell' - (i+1) + 1 \geq 1$ active rows. Let $J^{i+1} = M(i+1)^{i+1}$ be the $(i+1)$th column of $M(i+1)$. Define $m_{i+1} = choice(J^{i+1})$. Then m_{i+1} is an index of an active row (as $M_p[q] \neq *$ implies that M_p is an active row).

Observe that if $\ell^* \leq \ell$ then $\ell' = \ell^*$ is the number of active rows of M and every index of an active row is chosen, namely $\{m_1, \ldots, m_{\ell'}\}$ is the set of all indices of active rows. Hence we get:

Lemma 7.2 *If there exists an index of an active row that is not among the chosen indices $\{m_1, \ldots, m_{\ell'}\}$ then $\ell^* > \ell$ and so $\ell' = \ell$.*

Proof: If $\ell' \neq \ell$ then $\ell' = \ell * < \ell$, as $\ell' = \min\{\ell, \ell *\}$. But then every index is chosen. ■

Definition 7.3 *If $S = \langle (M(i), m_i) \mid 1 \leq i \leq \ell' \rangle$ is a sequence of matrices and pairwise distinct indices, then $\pi(S) = \langle m_i \mid 1 \leq i \leq \ell' \rangle$ is the sequence of indices alone, and for any index k, $\pi_k(S)$ is defined as follows. If k is not in $\{m_1, \ldots, m_{\ell'}\}$ then $\pi_k(S) = \pi(S)$, but if $k = m_j$ (for a unique j) then $\pi_k(S) = \langle m_i \mid i \leq j \rangle$.*

So we have observed in the lemma above that if j is an index of an active row of M and $j \notin \pi(\ choice(M)\ )$, then the length of $choice(M)$ is ℓ.

Definition 7.4 *Let M and F be two nonempty $N \times \ell$ ord-matrices, and*

$$choice(F) = \langle\ (F(i), f_i) \mid 1 \leq i \leq i_0\rangle$$

$$choice(M) = \langle\ (M(i), m_i) \mid 1 \leq i \leq j_0\rangle.$$

We say that M is a conservative extension of F if

1. M extends F. (Which means that $M_i = F_i$ whenever F_i is active.)
2. $i_0 \leq j_0$ and for every $k \leq i_0$ $(M(k))^k$ is a conservative extension of $(F(k))^k$.

Lemma 6.1 can be extended to the present context as follows.

Lemma 7.5 *1. If F is an $N \times \ell$ ord-matrix, then there is a fully defined (i.e., no $*$ entry) ord-matrix M that is a conservative extension of F.*

2. Suppose that an ord-matrix M is a conservative extension of an ord-matrix F. Then:

(a) $\pi($ $choice(F)$ $)$ is an initial segment of $\pi($ $choice(M)$ $)$. That is, $f_k = m_k$ for every $k \leq i_0$.

(b) For every index j of an active row of F, $j \in \pi($ $choice(M)$ $)$ iff $j \in \pi($ $choice(F)$ $)$.

3. If F_1, F_2, F_3 are ord-matrices such that F_2 extends F_1 and F_3 extends F_2, if F_3 is a conservative extension of F_1, then F_3 is a conservative extension of F_2 and F_2 is a conservative extension of F_1.

4. A conservative extension of a conservative extension of F is a conservative extension of F.

Proof: We first prove (1). Suppose that $f*$ is the number of active rows of F, $i_0 = \min\{\ell, f*\}$, and

$$choice(F) = \langle (F(i), f_i) \mid 1 \leq i \leq i_0\rangle.$$

We define M^i for $i \leq i_0$ as a conservative extension of $F(i)^i$. For $i_0 < i \leq \ell$, let M^i be any extension of F^i, obtained, say, by adding zeros to the undefined entries. It is quit clear that M is a conservative extension of F.

Now suppose that M is a conservative extension of F, and let $choice(F)$ and $choice(M)$ be as in Definition 7.4. To prove that $f_k = m_k$ for every $k \leq i_0$ we follow the definition of $choice(F)$ and the first i_0 steps in the definition of $choice(M)$. For $k \leq i_0$:

$$f_k = choice(F(k)^k),$$

$$m_k = choice(M(k)^k.$$

But column $M(k)^k$ is a conservative extension of $F(k)^k$, and hence Lemma 6.1(4) implies that $f_k = m_k$.

Next we prove item (2). If j is an index of an active row of F and $j \notin \pi(\ \mathit{choice}(F)\ )$ then $\pi(\ \mathit{choice}(F)\ )$ has length ℓ (by Lemma 7.2) and is thus equal to $\pi(\ \mathit{choice}(M)\ )$. So $j \notin \pi(\ \mathit{choice}(M)\ )$ as well.

The proof of items (3) and (4) is left to the reader. ∎

The function $\mathit{change\_order}(J, k)$, defined (in Definition 6.2) for ord-columns, is extended here to matrices.

Definition 7.6 *Let M be a nonempty ord-matrix and k an index. We are going to define the function $\mathit{change\_order}(M, k)$ which returns an active ord-row v.*

We follow the definition of $\mathit{choice}(M) = \langle (M(i), m_i) \mid 1 \le i \le \ell' \rangle$. For $1 \le i \le \ell'$ define

$$v[i] = \mathit{change\_order}(\ (M(i))^i, k)$$

and for $\ell' < i \le \ell$ define $v[i] = 0$.

Figure 10 illustrates some of the definitions introduced so far.

$$F = F(1) = \begin{pmatrix} 1 & 0 \\ * & * \\ 0 & 1 \\ * & * \end{pmatrix}, \quad F(2) = \begin{pmatrix} 1 & 0 \\ * & * \\ * & * \\ * & * \end{pmatrix}.$$

$$M = M(1) = \begin{pmatrix} 1 & 0 \\ * & * \\ 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad M(2) = \begin{pmatrix} 1 & 0 \\ * & * \\ * & * \\ 0 & 0 \end{pmatrix}.$$

Figure 10: M is a conservative extension of F. $\mathit{choice}(M) = \langle \langle M(1), 3\rangle, \langle M(2), 1\rangle \rangle$. $\pi(\mathit{choice}(M)) = \langle 3, 1\rangle$. $\mathit{change\_order}(M, 1) = \langle 1, 1\rangle$, $\mathit{change\_order}(M, 3) = \langle 1, 0\rangle$, $\mathit{change\_order}(M, 4) = \langle 0, 0\rangle$.

Now that we have defined the data structures and the operations on them we are ready for the algorithm itself which is explained in the following subsection.

```
repeat forever

    remainder section;
    isolation algorithm;
```

Figure 11: Higher-level view of the isolation protocol.

```
isolation algorithm (by P_i)

begin
   i1 write(X_i := true);
   i2 fill(M);
   i3 if return() goto i1;
   i4 write(X_i := false);
   i5 v := change_order(M, i);
   i6 write(ORD_i := v);
end
```

Figure 12: The *isolation algorithm* for P_i. M is an ord-matrix, *return*() is an unspecified boolean function, *change_order*(M, i) is defined in Definition 7.6, and v is an ord-vector.

```
procedure fill(M)

forall j concurrently do
    begin
        read(x_j := X_j);
        if x_j read(M_j := ORD_j) else M_j := the inactive row
    end
```

Figure 13: The *fill* procedure. M is an ord-matrix.

7.1 The isolation algorithm and its stabilization

Each process P_i executes the code of Figure 11. The remainder section may be terminating or nonterminating. However, an active process in its remainder section ensures that its X_i register is set to *false*. The *isolation algorithm* is described in Figure 12.

The *isolation algorithm* for P_i uses two regular registers:

1. X_i is boolean, and
2. ORD_i carries $\{0,1\}$-rows.

Initially P_i sets its X_i register (in line **i1**), and then it collects all ord-rows values from processes that were found with $X_j = true$ and assigns the resulting matrix to M. Procedure *fill*, executed by P_i, is defined in Figure 13. (In executing *fill*(M), P_i reads X_i as well (which has value *true*) and reads ORD_i.) Then the unspecified boolean function *return*() determines if to continue or to return the control to **i1**. If the goto loop (**i1**–**i3**) terminates, then **i4** is executed, and the function $change\_order(M, i)$ is applied to determine an active ord-vector v written in ORD_i. (Procedure *change_order* is defined in Definition 7.6.)

In order to formulate the properties of this algorithm we specify two types of processes (in any given system execution).

E0 A process is *frozen* if its X and ORD registers acquire a final, fixed value. Processes are said to be "frozen with $X = true$" if the value of their X register is (finally) fixed to be *true*. Frozen processes are further subdivided in three sub-types.

E0a Crashed processes are frozen.

E0b Active processes are frozen if they are (finally) in their remainder sections. We assume that an active process takes care of its registers, even when it is in the remainder section, so that if P_j is in its remainder section then its X_j register is (finally) constantly *false*, and its ORD_j register has some fixed value.

E0c An active process P_j that is (finally) locked in its goto loop is also frozen. Namely it gets (after a certain time) the value *return*() = *true* in each execution of **i3**. The ORD_j register of P_j never changes and its X_j register is fixed to *true*.

E1 An *unfrozen* process is one that is active and has always terminating goto loops. An unfrozen process executes an infinite number of *change_order* operations (line **i5**). (All frozen processes execute only a finite number of *change_order* operations.)

Processes of type E0c are frozen with $X = true$; processes of type E0b are frozen with $X = false$, and those of type E0a may have any frozen value for X.

Definition 7.7 *Let $\mathcal{S}$ be a system execution with at least one frozen process with $X = true$. The "frozen matrix" of $\mathcal{S}$ is the ord-matrix F defined by the following demands. Row i is active in F iff P_i is frozen with $X_i = true$, and F_i is the (final) constant value of ORD_j.*

Theorem 7.8 *Let F be the frozen matrix of $\mathcal{S}$. Then there is a moment τ such that if M is any ord-matrix obtained by executing* **i2** *(namely fill(M)) after τ, then M is a conservative extension of F.*

This explains the name *isolation algorithm* given to our algorithm. If there are any frozen processes with $X = true$, whose number is less than or equal to ℓ, then they are "isolated", in the sense that the active processes that execute *fill*(M) can be sure that the indices of these frozen processes are (finally) found in *choice*(M).

The proof relies on Theorem 6.3 which proves the corresponding result for the *single column algorithm.* It is clear that after all frozen processes have written their final values, any execution of *fill*(M) determines an ord-matrix M that extends F. The additional active rows, those in M that are not in F, correspond to active unfrozen processes. Let $\ell' = \min(\ell, f)$ and $choice(F) = \langle\ (F(i), f_i) \mid 1 \leq i \leq \ell'\rangle$. Now Theorem 6.3, or rather the arguments there, give that there is a moment τ_1 such that any execution of *fill*(M) after τ_1 determines that the first column M^1 is a conservative extension of F^1. So $choice(M^1) = choice(F^1) = f_1$. Hence when an active unfrozen process P_j executes $change\_order(M, j)$ it defines $M(2) = inactivate(M, f_1)$ in the second step. Now the argument of Theorem 6.3 applies to $M(2)^2$ and F^2; there is a moment such that all values of $M(2)^2$ are found to be conservative extensions of F^2. And so on, the argument goes for each $i \leq \ell'$ and gives that $M(i)^i$ is a conservative extension of $F(i)^i$.

8 The Combined Algorithm

The *combined algorithm* (Figures 14 and 15) is obtained by incorporating the *isolation algorithm* (Figure 12) into the *safe algorithm* (Figure 4). In particular, *read-all-TRY*() and *read-all-VEC*() are as in the *safe algorithm.* The function *getTry*(), unspecified in the *safe algorithm*, is now implemented by the code of Figure 15. Textually, the only difference between the *combined algorithm* and the *safe algorithms* is the addition of lines **c10, c11, c12** (and the definition of procedure *getTry*()). Observe that these lines and the call to *getTry*() do not write on any register that the rest of the algorithm reads. Hence by omitting the events corresponding to **c10–c12** from any execution of the *combined algorithm*, and by lumping the events in any execution of *getTry*() into a single higher-level event, one obtains an execution of the *safe algorithm.* Hence, the self-stabilizing ℓ-exclusion property for the *safe algorithm* applies here and we get self-stabilizing ℓ-exclusion for the *combined algorithm.* This proves the Safety Requirement 2.2.

Given a system execution $\mathcal{S}$ containing read/write events on some registers, if H is a subset of the registers used, then the set of all events in $\mathcal{S}$ pertaining to H forms a system execution which is said to be a *projection* of $\mathcal{S}$. For any system-execution of the *combined algorithm* we can identify a projection which results in a system execution of the *isolation algorithm.* Take only the events corresponding to lines 1 and 3 in executions of procedure *getTry*(), and take the events corresponding to lines **c10–c12**. In other words take the events corresponding to registers X and ORD. For any process P_i the resulting events have the form described in Figure 16. This is just the isolation algorithm in which the call to the function *return*() is interpreted as an evaluation of condition not *try* or $\mid B \mid > \ell$. Thus the *isolation algorithm* is executed in any system execution of the *combined algorithm*, and the results of Section 7

The *combined algorithm* (by P_i)

begin
c1 $try := false$;
c2 write($TRY_i := false$);
c3 repeat
c3.1 $read\text{-}all\text{-}VEC(v)$;
c3.2 $report(v)$; (* variable *vector* is determined *)
c3.3 $old\text{-}try := try$;
c3.4 $try := getTry()$;
c3.5 write($TRY_i := try$);
c3.6 if try & $\neg old\text{-}try$ write($VEC_i := vector$);
c4 until try;
c5 concurrently do
{ $read\text{-}all\text{-}VEC(v)$, $read\text{-}all\text{-}TRY(t)$}
c6 Define the set of indices B as follows
$j \in B$ if and only if t_j & $\neg(v_j$ dominates $v_i)$
c7 if $\mid B \mid > \ell$ goto **c3**;
c8 CS;
c9 write($TRY_i := false$);
c10 write($X_i := false$);
c11 $v := change\_order(M, i)$;
c12 write($ORD_i := v$);
end

Figure 14: The *combined algorithm.*

apply here. In particular, we shall use Theorem 7.8 there. What is the intuition behind the definition of A in line 5 of $getTry()$?

$$j \in A \equiv j \neq i \ \& \ [j \in \pi \ \vee \ (t_j \ \& \ \exists(i' \neq i) \ (i' \in \pi \ \& \ \neg(v_j \text{ dominates } v_{i'})))]$$

The set π, excepting i contains all indices j that the isolation process found to have higher priority than than i. In addition to this set, if process j does not dominate a process i' in π, then j is considered to be possibly behind i', and so j itself seems to be older than i and so to have a higher priority. thus A is the set of all processes found to have higher priority, and if this set has cardinality at least ℓ then variable *try* is set to *false* and process i resigns and write *false* on TRY_i.

Let $\mathcal{S}$ be any system execution of the *combined algorithm* in which the starting state of each process is arbitrarily determined. The registers and local variables can have any possible function $getTry()$

1. write($X_i := true$);
2. $read\text{-}all\text{-}TRY(t)$;
3. $fill(M)$;
4. $\pi := \pi_i(\ choice(M)\ )$;
5. $j \in A \equiv j \neq i \ \&\ [j \in \pi \vee (t_j \ \&\ \exists(i' \neq i)\ (i' \in \pi \ \&\ \neg(v_j \text{ dominates } v_{i'})))]$;
6. return $\mid A \mid < \ell$;

Figure 15: Function $getTry()$ by P_i.

1. repeat
   (a) write($X_i := true$)
   (b) $fill(M)$

   while not try or $(\mid B \mid > \ell)$
2. write($X_i := false$)
3. $v := change\_order(M, i)$
4. write($ORD_i := v$)

Figure 16: The *combined algorithm* contains the *isolation algorithm*.

values, and the control of each process is located in an arbitrary position. Some of the processes are crashed, which means that they are inactive, and the other processes are active forever (but some may be in the remainder section). The processes are classified as follows.

F0 The crashed processes are of type F0 if their TRY or X register is set to *true*.

F1 A process is of type F1 if its TRY and X registers are (finally) *false* forever. Thus active processes that are in their remainder section forever are of type F1. In addition, crashed process may be of type F1 (if they are not of type F0).

F2 Active processes that are (finally) locked in the goto loop (**c3**–**c7**) are of type F2. Processes of type F2 are further divided into

F2a those of type F2a which determine infinitely often $try = false$ in executions of **c3.4**, and

F2b those of type F2b which (finally) obtain $try = true$ at each execution of **c3.4**.

Processes of type F2 do not (finally) change their ORD registers, and their X register is (finally) set to $true$ (by $getTry()$).

F3 Active processes that have only successful goto loops, and execute an infinite number of times their algorithm are of type F3. Thus a process of type F3 executes *change_order* an infinite number of times. Of course, none of the registers of a process of type F3 is frozen.

Processes of types F0, F1, F2 are called frozen since (in some final segment) they never change the values of registers X and ORD.

By extension we say that an index i is of type F0 (or F1, F2, F3) or is frozen if P_i is of type F0 etc.

Our aim is to prove the Liveness Requirement 2.3 in $\mathcal{S}$. So we assume that the number of crashed processes is strictly less than ℓ. We must prove that each active process eventually executes its critical section, namely we must prove that there are no processes of type F2. Assume towards a contradiction that there is such a process. Let Y be the set of crashed indices, i. e. indices of crashed processes. Let Z be the set of frozen indices with $X = true$. Then $Y_X = Y \cap Z$ is the set of crashed indices that have their X register fixed to $true$. By assumption, $\mid Y \mid < \ell$, and hence $\mid Y_X \mid < \ell$. $Z \setminus Y$ is the set of indices that are frozen with $X = true$ but yet not crashed, namely processes of type F2. By assumption $Z \setminus Y \neq \emptyset$ and hence $\mid Y_X \mid < \mid Z \mid$.

Let F be the frozen matrix of $\mathcal{S}$, that is, the ord-matrix with active rows formed from ORD_k vectors for $k \in Z$. So F has $\mid Z \mid$ active rows.

Let $\ell' = \min\{\ell, \mid Z \mid\}$. Since $\mid Y_X \mid < \mid Z \mid$ and $\mid Y_X \mid < \ell$, $\mid Y_X \mid < \ell'$.

Let

$$choice(F) = \langle (F(k), f_k) \mid 1 \leq k \leq \ell' \rangle, \text{ and}$$

$$\pi(\ choice(F)\ ) = \langle f_1, \ldots, f_{\ell'} \rangle.$$

As $\mid Y_X \mid < \ell'$, there is in $\{f_1, \ldots, f_{\ell'}\}$ an index not in Y_X, and we define i_0 to be the first such index. That is $i_0 = f_k$ is active and for every $1 \leq j < k$ $f_j \in Y_X$ is crashed. Since $f_1, \ldots, f_{\ell'}$ are all in Z, $i_0 \in Z \setminus Y_X$ is of type F2.

Consider now process P_{i_0}. It is of type F2, and we first prove that it is of type F2b, namely TRY_{i_0} is frozen with value $true$.

Let τ be the moment given by Theorem 7.8. Namely, whenever $fill(M)$ is executed after τ, M is a conservative extension of F. Hence the choice sequence of M agrees with F that $f_1, \ldots, f_{\ell'}$ are the first ℓ' indices.

We take this moment τ to be sufficiently far so that every P_j of type F3 dominates (after τ) any crashed process. It suffices that P_j executes two rounds of the *combined algorithm* after the crash. Moreover, we may assume that any frozen process has already written its frozen value before τ.

Let e_0 be any execution of *getTry*() by P_{i_0} after τ, and consider the values π^{e_0} and A^{e_0} obtained in e_0 in executing lines 4 and 5. Then $\pi^{e_0} = \{f_1, \ldots, f_k\}$. We shall prove that $\mid A^{e_0} \mid < \ell$, and hence that P_{i_0} is of type F2b.

Claim 8.1 *For any j, if P_j is of type F3 or F1 then $j \notin A^{e_0}$.*

Proof: If P_j is of type F3 then $j \notin \pi^{e_0}$ (j is not in $\{f_1, \ldots, f_{k-1}\}$ because these processes are all crashed and j is not crashed. Clearly $j \neq f_k$ because f_k is of type F2).

In order to prove that $j \notin A^{e_0}$ it remains to prove that the following condition taken from the definition of A does not hold

$$\exists (i' \neq i)(i' \in \pi \ \& \ \neg(v_j \text{ dominates } v_{i'}))$$

Namely we want to prove the negation of

$$\exists f \in \{f_1, \ldots, f_{k-1}\} \ \neg(v_j \text{ dominates } v_f)$$

Yet P_j is of type F3, and hence v_j dominates all crashed processes and in particular v_j dominates all v_f.

An index j of type F1 is not in A^{e_0} since processes of type F1 have their X and *TRY* registers fixed to *false*. ■

It follows from this claim that A^{e_0} consists of indices of type F0 and F2. As we shall see, the protocol is designed to make $\mid A^{e_0} \mid < \ell$ by forcing processes of type F2 to renounce their *TRY* variables (there can be no control over the crashed process of course).

Define next

$$Y_0 = Y \cap A^{e_0}.$$

That is, Y_0 consists of the crashed process in A^{e_0}. As $Y_0 \subseteq Y$, $\mid Y_0 \mid < \ell$. Since Y_0 is the set of crashed indices in A^{e_0}, the previous claim implies that $Y_1 = A^{e_0} \setminus Y_0$ consists of indices of type F2. If $j \in Y_1$ then $t_j = \textit{true}$ holds in e_0, because $j \notin \pi$ in e_0. Following the definition of A we find an explicit definition for Y_0:

$$Y_0 = \{f_1, \ldots, f_{k-1}\} \cup \{j \in Y \mid Q(j)\} \tag{3}$$

Where $Q(j)$ is the following statement:

$$P_j \text{ is frozen with } \textit{TRY}_j = \textit{true} \ \& \ \exists k' < k \ \neg(P_j \text{ dominates } P_{f_{k'}}). \tag{4}$$

Since $Y_0 \cup \{f_k\}$ has $\leq \ell$ indices and $Y_0 \cup \{f_k\} = Y_0 \cup \{f_1, \ldots, f_k\}$, we can make the following definition.

Let $m_0 \leq \ell'$ be the maximal index such that

1. $k \leq m_0 \leq \ell'$, and
2. $Y_0 \cup \{f_1, \ldots, f_{m_0}\}$ contains $\leq \ell$ indices.

We shall prove that

$$A^{e_0} \subseteq Y_0 \cup \{f_1, \ldots, f_{m_0}\}, \tag{5}$$

and hence that $\mid A^{e_0} \mid < \ell$ (since $i_0 = f_k \notin A^{e_0}$, item 2 implies $\mid A^{e_0} \mid < \ell$). So suppose that $j \in A^{e_0}$. Assuming that $j \notin Y_0$, we shall deduce that $j \in \{f_1, \ldots, f_{m_0}\}$. So $j \in A_1 = A^{e_0} \setminus Y_0$ is of type F2, and as $t_j = true$ necessarily holds in e_0 the following can be employed to conclude (5).

Claim 8.2 *If P_j is of type F2 and $j \notin \{f_1, \ldots, f_{m_0}\}$, then P_j is frozen with $TRY_j = false$.*

Proof: Any process of type F2 is frozen with $X = true$, but we must show that P_j renounce its tries and freezes with $TRY_j = false$. Look at an execution e of *getTry*() by P_j occurring after τ. Let M^e, π^e, A^e be the values of these variables in e. M^e is a conservative extension of the frozen matrix F and hence $\pi($ *choice*(M^e) $)$ begins with $f_1, \ldots, , f_{\ell'} = \pi($ *choice*(F) $)$ (Lemma 7.5 (1)). Then $\pi^e = \pi_j(choice(M^e))$.

Consider first the case that $j \notin \pi^e$. Then π^e is of length ℓ (since j is active in M^e, and by Lemma 7.2). In this case $\pi^e \subseteq A^e$ implies that e returns *false*.

Now suppose that $j \in \pi^e$ and then j is the last index of π^e. Since j is of type F2, j is frozen with $X = true$ and as $j \in \pi($ *choice*(M^e) $)$ $j \in \pi($ *choice*(F) $)$ as well (by Lemma 7.5(2)). Say $j = f_{m'}$. Yet $j \notin \{f_1, \ldots, f_{m_0}\}$, by our assumption, and hence $m' > m_0$. But then the definition of m_0 implies that $Y_0 \cup \{f_1, \ldots, f_{m'}\}$ contains $> \ell$ indices, and so $Y_0 \cup \{f_1, \ldots, f_{m'-1}\}$ contains $\geq \ell$ indices. Yet Equation (3) implies that $Y_0 \subseteq A^e$. So $Y_0 \cup \{f_1, \ldots, f_{m'-1}\} \subseteq A^e$, and *getTry*() in e returns *false* in this case as well. ■

We have proven that P_{i_0} stabilizes with $TRY_{i_0} = true$. Hence the repeat-until loops are always successful and the writes in VEC_{i_0} in line **c3.6** are (finally) never executed, and so VEC_{i_0} stabilizes at some fixed value. This enables all process of type F3 to dominate P_{i_0}. Let τ_1 (after τ) be a moment after which all write on VEC by processes of type F3 dominate the stabilized write by P_{i_0}. We now argue *after* τ_1, and we shall prove that the goto loop for P_{i_0} must be successful, which is the desired contradiction.

It follows that in calculating B (in any sufficiently late execution of line **c6** by P_{i_0}) indices of type F3 are all excluded. Indices of type F1 are certainly not in B, because they freeze with $TRY = false$. Hence B contains only indices of type F0 and F2. Let B_0 be the set of indices of B of type F0. That is, $B_0 = B \cap Y$. $m \in B_0$ iff P_m is crashed with $TRY_m = true$ and P_m does not dominate P_{i_0}.

Let $m_1 \leq \ell'$ be the maximal index such that $B_0 \cup \{f_1, \ldots, f_{m_1}\}$ contains $\leq \ell$ indices. m_1 is well defined and in fact $m_1 \geq k$, since $B_0 \cup \{f_1, \ldots, f_{k-1}\} \subseteq Y$ (by minimality of $i_0 = f_k$) and it hence has cardinality $< \ell$.

We shall prove that

$$B \subseteq B_0 \cup \{f_1, \ldots, f_{m_1}\}$$

and hence that $\mid B \mid \leq \ell$. This shows that line **c7** crosses successfully, and hence that i_0 is not of type F2! The proof of this inclusion is similar to the previous claim. If $j \in B$ and $j \notin B_0$ then j is of type F2, and hence the following claim suffices.

Claim 8.3 *If P_j is of type F2, and $j \notin \{f_1, \ldots, f_{m_1}\}$, then P_j freezes with $TRY_j = false$ (and so $j \notin B$).*

Proof: Look at some execution e of *getTry*() by P_j of type F2, such that $j \notin \{f_1, \ldots, f_{m_1}\}$. Then M^e (obtained in e in executing *fill*(M)) is a conservative extension of F. Assume first that $j \notin \pi_j($ *choice*(M^e) $)$. Then π^e is of length ℓ, and e returns *false* since $\pi^e \subseteq A^e$. So assume next that $j \in \pi^e$ and hence that $j \in$ *choice*(F), and $j = f_{m'}$. By our assumption $m' > m_1$ and hence $B_0 \cup \{f_1, \ldots, f_{m'}\}$ contains more than ℓ indices and $B_0 \cup \{f_1, \ldots, f_{m'-1}\}$ contains at least ℓ indices. But this implies, as we show below, that A^e contains $\geq \ell$ indices and hence that e returns *false*, as desired. We prove that

$$B_0 \cup \{f_1, \ldots, f_{m'-1}\} \subseteq A^e.$$

First $f_1, \ldots, f_{m'-1}$ are all in A^e, because they satisfy the clause "$j \in \pi$" in line 5. Now if $m \in B_0$ then the frozen value of VEC_m does not dominate P_{i_0} (this follows the definition of B in c6). Moreover, $TRY_m = true$ by definition of B_0. Hence $m \in A^e$. Indeed, in defining A^e, t_m is true and the existential formula

$$\exists (i' \neq i) \ (i' \in \pi \ \& \ \neg(v_m \text{ dominates } v_{i'}))$$

holds in e by virtue of $i' = i_0$. ∎

We can finally state our result.

Theorem 8.4 *The combined algorithm is self-stabilizing ℓ-exclusion algorithm.*

Proof: The safety requirement is proved in Lemma 5.1, and the no-starvation requirement follows from Claim 8.3 that proves that no process of type F2 exists, and hence requirement 2.3 holds. ∎

9 Discussion

We have presented here for the first time a self-stabilizing protocol for the ℓ-exclusion problem. Self stabilization was first defined by Dijkstra who based this notion on states. Our protocol uses regular registers, which were defined by Lamport by means of system executions. Consequently we had to define a new model which combines these two approaches. Several new ideas and techniques were employed in the design of the algorithm. In particular we believe that the isolation algorithm (which selects process identifiers to achieve fairness) may find more applications.

Future work will address stronger notions of fairness, such as first come first enabled (see [4]). Another interesting path for future investigation is to reduce or even optimize the resources (time and memory) required for the solution.

Acknowledgment: It is a pleasure to thank Jim Burns for his helpful remarks.

References

[1] U. Abraham, "On interprocess communication and the implementation of multi-writer atomic registers" *Theoretical computer science,* Vol. 149, pp. 257–298, 1995.

[2] U. Abraham, *Models for Concurrency*, Gordon and Breach, 1999.

[3] U. Abraham, S. Dolev, T. Herman and I. Koll, "Self-Stabilizing ℓ-Exclusion," in *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pp. 48–63, Carleton University Press, 1997.

[4] Y. Afek, D. Dolev, E. Gafni, M. Merritt and N. Shavit, "A Bounded First-In, First-Enabled Solution to the ℓ-exclusion Problem," *ACM Transactions on Programming Languages and Systems,* Vol. 16, No. 3, pp. 939-953, 1994.

[5] H. Attiya, J. L. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, McGraw-Hill, 1998.

[6] E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM,* Vol. 17, No. 11, pp. 643-644, 1974.

[7] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communication of the ACM,* Vol. 8, No. 9, p. 569, 1965.

[8] S. Dolev, A. Israeli and S. Moran, "Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity", *Distributed Computing*, Vol. 7, pp. 3-16, 1993. *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, Montreal, August 1990, pp. 103-117. *Proc. of the first Workshop on Self-Stabilizing Systems*, 1989.

[9] S. Dolev, *Self-Stabilization*, to appear in MIT Press, 2000.

[10] D. Dolev, and N. Shavit, "Bounded Concurrent Time-Stamping" *SIAM Journal on Computing*, Vol. 26, No. 2, pp. 418-455, 1997.

[11] C. Dwork, M. Herlihy, S. Plotkin and O. Waarts, "Time lapse snapshots," in *Proceedings of the Israel Symposium on the Theory of Computing and Systems,* Springer-Verlag LNCS 601, pp. 154-170, 1992.

[12] M. Flatebo, A. K. Datta, and A. A. Schoone, "Self-stabilizing multi-token rings," *Distributed Computing,* Vol. 8, pp. 133-142, 1994.

[13] M. Fischer, N. Lynch, J. Burns and A. Borodin, "Resource allocation with immunity to limited process failure," *Proceedings 20th IEEE Annual Symposium on Foundations of Computer Science*, pp. 234-254, 1979.

[14] M. Fischer, N. Lynch, J. Burns and A. Borodin, "Distributed FIFO allocation of identical resources using small shared space," *ACM Transactions on Programming Languages and Systems,* Vol. 11, pp. 90-114, 1989.

[15] A. Israeli and M. Li, "Bounded time stamps," *Proc. of the 28th Annual Symposium on Foundations of Computer Science*, pp. 371-382, 1987.

[16] A. Israeli and M. Li, "Bounded time stamps," *Distributed Computing,* Vol. 6, pp. 205-209, 1993.

[17] A. Israeli and M. Pinchasov, "A linear time bounded concurrent timestamp scheme," Technical Report, Technion, Haifa, Israel, March 1991.

[18] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Communication of the ACM,* Vol. 17, No. 8, pp. 453-455, 1974.

[19] L. Lamport, "The mutual exclusion problem. Part II: Statement and solutions," *Journal of the ACM*, 33(2):327-348, 1986.

[20] L. Lamport, "On interprocess communication," (Parts I and II), *Distributed Computing 1, 2*, 1986, 77-101.

[21] E. A. Lycklama and V. Hadzilacos, "A First-Come-First-Served Mutual-Exclusion Algorithm with Small Communication Variables," *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, 1991, pp. 558-576.

[22] N. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.

[23] G. Peterson, "Myths About The Mutual Exclusion Problem," *Information Processing Letters,* Vol. 12, No. 3, pp. 115-116, 1981.

[24] G. Peterson and M. Fischer, "Economical Solutions for the Critical Section Problem in Distributed Systems," In *Proceedings of the 9th ACM Symposium on Theory of Computing,* pp. 91-97, 1977.

[25] N. Wiener, "A contribution to the theory of relative position," *Proc. Camb. Philos. Soc.*, vol. 17, pp. 441-449, 1914.

[26] M. Raynal, *Algorithms for Mutual Exclusion*, MIT Press, 1986.

[27] K. Truuvert, M.S. thesis, Dept. of Computer Science, Univ. of Toronto, Ontario, 1989.