Proceedings of the Fifth Jerusalem Conference on Information Technology, Jerusalem, Israel, October 1990
IEEE Computer Society Press, Los Alamitos, CA, pp. 110-117

Mutual Exclusion Revisited†

Boleslaw K. Szymanski

Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180

Abstract

A family of four mutual exclusion algorithms is presented. Its members vary from a simple three-bit linear wait mutual exclusion to the four-bit first-come first-served algorithm immune to various faults. The algorithms are based on a scheme similar to the Morris's solution of the mutual exclusion with three weak semaphores. The presented algorithms compare favorably with equivalent published mutual exclusion algorithms in their program's size and the number of required communication bits.

1. Introduction

Mutual exclusion is at the center of many concurrent process synchronization problems and, consequently, is of a great theoretical and practical significance in parallel and distributed processing. In the mutual exclusion problem, there is a collection of asynchronous processes. Each process contains a distinct part of the code called a *critical section (or region).* The process's remaining code is referred to as a *noncritical section (or region)* [2]. Each process alternately executes its noncritical and critical sections. Processes can proceed in parallel outside of the critical section but only one process at a time can execute the critical section.

Mutual exclusion in uniprocessor systems can be provided by disabling interrupts when a process is in its critical section. Such a solution is efficient only if critical sections are short. Otherwise the system response time would degrade and disabled interrupts could be mishandled. The other limitation of this technique is that in most systems interrupt disabling and enabling is beyond control of the user programs.

† This work was partially supported by the National Science Foundation under grant No. CCR-8613353 and by the Army Research Office under contract DAAL03-86-K-0112

In multiprocessors with a shared memory, a special test-and-set instruction can be used to support the mutual exclusion. However, this solution requires synchronized accesses to the shared memory from all processes and such accesses could be difficult to support. In a multiprocessor multiport memory system the test-and-set instruction cannot be implemented by controlling an access cycle of a single processor [4], [11]. On a large VLSI chip processors cannot run on the same clock because sending a clock pulse across the chip introduces a delay in a pulse propagation. Growing popularity of parallel and distributed architectures has led to renewed interest in algorithmic solutions to the mutual exclusion problem [1], [4], [6], [7], [9], [11], [12], [13].

Algorithmic solutions to the mutual exclusion problem were extensively studied in the past [2], [3], [5], [12]. Recently, Lamport in [7] presented a new extended definition of the mutual exclusion and its four solutions characterized by different degrees of enforced fairness and robustness. Lamport's algorithms are immune to several types of process malfunctions. Unlike the majority of older solutions, his algorithms do not assume that read/writes from/to communication variables are mutually exclusive. Such robustness is important in large distributed systems where failure of a single processor should not break down the entire system. It is also needed in VLSI chip based multiprocessor systems, in which nonuniform conditions in the chip's wafer result in varying reliability of individual processors.

In Lamport's algorithms, the desired degree of fairness and robustness decides the number of communication variables required by each process. Let n denotes the number of processes participating in the mutual exclusion. The strongest fairness condition (known as first-come first-served property) together with the strongest robustness requirement are provided by the algorithm that uses n-factorial of communication binary variables per process. The fair solution with a constant number of communication variables was published in [13] (linear wait, four one-bit communication variables), and reported in [8] (first-come first-served, five one-bit

communication variables) and in [14] (first-come first-served, four one-bit communication variables). The algorithm with 17 bit communication variables immune to all types of malfunctions defined by Lamport was presented in [15] . The author of that report conjectured:

> "Unfortunately, the presented algorithm is quite long and complex... It remains to be seen whether a shorter and simpler algorithm with the same properties exists. A conjecture is made that if a shorter solution exists, it will be of the same basic structure; that is, part of the algorithm will be constructed from a weaker solution to the same problem, along with a local critical section."

The algorithms presented in this paper show that these requirements were too strict. What suffices is a "separation" algorithm which keeps the processes eligible for mutual exclusion separate from newly arriving processes (see four-bit robust algorithm in section 3). Our four-bit (and about a quarter of the size of the program in [15]) self-stabilizing first-come first-served algorithm includes only one three-bit self-stabilizing algorithm with linear wait as a basic component.

The algorithms presented in this paper are based on a scheme similar to the Morris's solution of the mutual exclusion with three weak semaphores [10].

2. The Problem Statement

The Lamport's definition of mutual exclusion has been presented in [7], so only a general description is given here, following also [13]. There are n (n>1) processes that are numbered from 0 to n-1. The processes are executing independently of each other, possibly on different processors. Each process contains a portion of the code called a *critical section*, which often includes accesses to limited resources. The rest of the process code is called a *noncritical section.* There is no assumption about the rate at which processes execute. However, each process in its critical section makes a finite progress. This means that a finite, but possibly unbounded, amount of time elapses between the execution of individual instructions of the code. In addition, it is assumed that a process entering its critical section will leave it after a finite amount of time.

Each process starts its execution at a specified location in the noncritical section with all variables set to initial values. Processes alternately execute their noncritical and critical sections. A process may enter its critical section any number of times. Processes can communicate with each other through *communication variables.*

Algorithmic solutions to the mutual exclusion problem consists of two sections of code that surround the critical section in each process. The first section is executed before the critical section and is called a *prologue* or *trying.* The second section is executed after the critical section, and is called an *exit.* The assumption about the finite progress in execution of the critical section is extended to prologue and exit sections as well. However, the extended assumption does not imply that a process which started to execute its prologue or exit section has to leave any of them in a finite time. In other words, an infinite looping is not excluded by assumption and should be avoided through proper design of the algorithm. We are interested in a uniform solution, in which the prologue and exit sections are the same in each process.

There are four properties required from the solution.

I. **Mutual exclusion:** For any pair of distinct processes, any two of their critical section executions are disjoint in time.

II. **Deadlock freedom:** If there is a nonterminating prologue section execution, then there are unbounded number of critical section executions. In other words, the critical section should never become inaccessible to all processes. If a number of processes attempt to execute their critical sections, then after a finite amount of time some of them should be able to do so.

III. **Fairness** (lockout freedom property): Every prologue section execution must terminate, i.e. no process will be denied entry to its critical section forever. The strongest fairness property is known as first-come first-served. For the purpose of defining this property, we assume that the prologue section consists of two parts: a *gate* section that requires executing only a bounded number of elementary operation (therefore a gate section always terminates), followed by a *waiting* section. The first-come first-served property is satisfied if for any pair of processes the following implication holds:

> if a gate section execution of one process is followed† by a gate section execution of the other, than the corresponding critical section executions of these processes are in the same relation.

Less restrictive fairness property is know as the linear wait. It requires that no process will enter its critical section twice while another process is waiting.

† i.e. the execution of the first gate section terminates before the execution of the other one starts.

IV. **Robustness:** The solution should be immune to the following types of malfunctions:

- **flickering bits** (read errors during writes): a read of the communication variable which is being concurrently written upon may return a random value. As pointed out in [11] read errors during writes can easily occur when two processors communicate while running under control of different clocks. The sum of pulses from two different processors may create so called runt pulse, which causes a read to return a random value.
- **shut-down** (premature termination): at any point of its execution, a process can reset† its communication variables and halt (shut-down represents the physical situation of unplugging a processor),
- **abortion:** at any point in its execution, a process can reset a predefined subset of its communication variables and then start executing again in its noncritical section,
- **failure:** a process keeps setting its state, including the values of its variables, to arbitrary values within the program's and variables' ranges and then aborts, never again malfunctioning,
- **transient malfunction:** a process keeps setting its state, including the values of its variables, to arbitrary values within the program's and variables' ranges and then resumes normal execution at any point in its program, never again malfunctioning.

The robustness requirements imply that processes can use only process specific communication variables [12]. Such variable can be written only by one process ("owner" of that variable). It may be read by all processes.

3. The Algorithms

The first algorithm presented in Figure 1 provides the mutual exclusion with linear wait. It uses three one-bit communication variables in each process, and is immune only to the flickering bits malfunctions. It is a modification of the algorithm presented in [13]. This algorithm is used as a building block for other algorithms presented in the paper.

† Since any write to a communication variable is not a null operation and can affect concurrent reads, "resetting a variable" means here assinging a default value to the variable only if at this instance the variable has a value different from the default.

The idea behind the algorithms is simple. The prologue section (statements p1-9 in Figure 1) simulates a waiting room with a door. All processes requesting entry to the critical section at roughly the same time gather first in the waiting room. Then, when there are no more processes requesting entry, processes inside waiting room shut the door and move to the exit from the waiting room. From there, one by one, they enter their critical sections in the order of their numbering. Any process requesting access to its critical section at that time has to wait in the initial part of the prologue section (at the entry to the waiting room).

The door to the waiting room is initially opened. The door is closed when a process inside the waiting room does not see any new processes requesting entry. The door is opened again when the last process inside the waiting room leaves the exit section of the algorithm.

Three one-bit process specific communication variables, called a (active, competing for a critical section), w (waiting inside the waiting room) and s (shutting the door to the waiting room), respectively, describe the status of a process. Each process can be in one of the following five states:

1) passive - all three communication variables are false (aws=false,false,false). A process in the passive state is executing the noncritical section.
2) entry - only the variable a is set to true (aws=true,false,false). A process in the entry state wants to access its critical section and attempts to enter the waiting room.
3) inside - only the variable w is set to true (aws=false,true,false). A process in the inside state passed through the door into the waiting room.
4) transient - two variables: s and w are set to true (aws=false,true,true). A process in the transient state shuts the door into the waiting room temporarily.
5) exit - only the variable s is set to true (aws=false,false,true). A process in the exit state keeps the door into the waiting room shut for good and is either executing its critical section or waiting for its turn to execute it.

A transition from the passive state to the entry state is unconditional. A process is allowed to move from the entry state to the inside state if the variable s is set to false in all processes (in other words, the door is not shut either temporarily or permanently). A process in the inside state that notices that there are no processes in the entry state (i.e. the variable a is set to false in each process) can move to the transient state and shut the

door temporarily by setting its variable s to true (statements p5-6). From the transient state a process checks again for presence of any processes in the entry state. If there are any, the checking process backs off to the inside state; otherwise it moves to the exit state and shuts the door for good. A process that reaches the exit state from the transient state will be called a *leader.*

```
communication variables::  a, w, s: boolean = false
private variables::        j: 0..n

p1:  ai=true;
p2:  for(j=0;j<n;j++) while(sj);
p3:  wi=true; ai=false;
p4:  while(!si) {
p5:     for (j=0;j<n & !aj;j++);
p6:     if (j==n) { si=true;
p6.1:      for (j=0;j<n & !aj;j++);
p6.2:      if (j<n) si=false;
p6.3:      else { wi=false;
p6.4:         for(j=0;j<n;j++) while(wj);
          }
        }
p7:     if (j<n) for (j=0;j<n & (wj | !sj);j++);
p8:     if (j!=i & j<n) {
p8.1:      si=true; wi=false;
        }
    }
p9:  for(j=0;j<i;j++) while(wj | sj);
    Critical Section
e1:  si=false;
```

Figure 1. Three-Bit Linear Wait Algorithm†

A leader waits in the exit state for processes in the inside state to move to the exit state too.

A process in the inside state that notices a process in the exit state moves to the exit state immediately (statements p7-8.1). From the exit state processes enter critical section in the order of their numbering (statement p9).

If a process in the inside state attempts and fails to become a leader (by moving to the transient state and backing off to the inside state) then there is a process that was in the passive state before that attempt and then moved to the entry state (compare loop in statement p6.1). Until the moved process reaches the inside state, the attempting process will not move to the transient state again. Since processes can leave the waiting room only by passing through the exit state and the leader waits for processes in the inside state to reach the exit state, then no process can leave the waiting room until the door is closed. It follows from the above that a process can make at most n-1 attempts to become a leader before executing its critical section. On the other hand, if none of the processes in the inside state attempts to become a leader, than there is a process in the entry state which can move to the inside state (since the door is not shut). When all processes in the entry state reach the inside state, each process in the inside state will be able to move to the transient state. Hence, after a finite time some process(es) will become leader(s).

If the highest numbered process in the waiting room is a leader, it will keep all processes in the entry state looping on its variable s (compare statement p2). Otherwise the highest numbered process inside the waiting room will set its variable s to true before any leader will enter critical section. Hence, each process in the entry state will loop at least to that moment on the variable s of one of the leaders and finally on the variable s of the highest numbered process. It follows that the processes in the entry state cannot reach the inside state from the moment any process reaches the exit state until all processes inside the waiting room leave their critical sections. In other words, our algorithm separates processes in the passive and entry states from processes in the exit, transient and inside states. The separation is achieved in a finite time and lasts from the moment a leader reaches the exit state until all processes already inside the waiting room leave their critical sections.

With the separation property demonstrated, showing that the presented algorithm enforces the mutual exclusion with linear wait is simple. At any time only one process can have the lowest order number in the set of processes that reached the exit state, so the mutual exclusion is enforced. No process will wait in the waiting room forever, since a leader is created in a finite time and then all processes in the inside state are able to reach the exit state directly. No process will wait in the exit state forever either. There is always a process that is either in its critical section or able to reach it. Thus, the algorithm is deadlock free. Finally, if a process leaves the critical section, then it cannot pass the door into the waiting room until all processes that waited with it inside the waiting room executed the critical section. Moreover, those and only those processes that reached the entry state before a leader causing the separation reached the transient state are inside the waiting room after the separation. Hence, if one process reaches the entry state before the other, then the next separation cannot leave the former process before the waiting room and the latter process inside the waiting room (all other combinations are possible, however). Thus, the linear wait is enforced.

It is possible to extend the presented above algorithm to obtain a solution that is immune to all malfunc-

† The author acknowleges help of Vladislavs Jahundovics in removing a typo from this program.

tions defined in section 2. In the basic algorithm only the leader needs to be delayed in the exit state until all processes in the transient and inside states reach the exit state. If the leader aborts or shut-downs before some processes in the inside state reach the exit state but after some other processes already executed the critical section then the linear wait requirement can easily be violated. Thus, in the robust algorithm presented in Figure 2 all processes that reached the exit state wait until processes in the inside and transient states move to the exit state (notice a new position of the loop in statement p6.4).

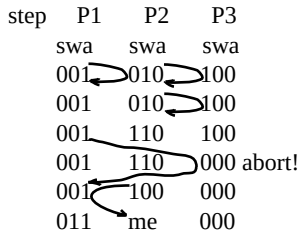
```
communication variables::  a, w, s: boolean = false
private variables::            j, k: 0..n

p1:  ai=true;
p2:  for(k=1;k<n;k++) for(j=0;j<n;j++)
p3:      while(sj && i<>j) { wi*=*false; si*=*false; }
p4:  while(!si) { wi*=*true; ai*=*false;
p5:      for (j=0;j<n & !aj;j++);
p6:      if (j==n) { si=true;
p6.1:        for (j=0;j<n & !aj;j++);
p6.2:        if (j<n) si=false;
p6.3:        else wi=false;
         }
p7:      if (j<n) for (j=0;j<n & (wj | !sj);j++);
p8:      if (j!=i & j<n) { si=true;
p8.1a:       if (!sj) si=false;
p8.1b:       else wi=false;
         }
     }
p6.4:for(j=0;j<n;j++) while(wj && i<>j) {ai*=*false;
wi*=*false;}
p9:  for(j=0;j<i;j++) while(sj && i<>j) wi*=*false;
     Critical Section
e1:  si=false;
```

Figure 2. Three-Bit Robust Linear Wait Algorithm

Even a process in the exit state may fail to keep the door closed†, if a shut-down or abortion takes place. In the scenario in Figure 3, a process P1 is able to sneak through the door into the waiting room, although in the view of a process P2 the door was closed by the variable s either in the process P3 or the process P2. Please note, that the second check of values of the communication variables would succeed in discovering that the door is indeed closed. In general, k+1-st check of communication variables yields the proper status of the door in the presence of at most k abortions and shut-downs of leader processes. Aborted and shut-down processes cannot pass through the door until the processes remaining inside the waiting room exit it. Thus, checking values of

† The author wish to thanks Prof. Amir Pnueli for pointing out this possibility.



Arrows show order of checking values of bit variables.

Figure 3. An Example of Possible ME Violation

the communication variables has to be done at most n-1 times to ensure the proper result (see an additional loop in statement p2 in Algorithm 2).

It is also necessary to prevent deadlock from occurring as the result of transient malfunctions. Since the unbounded waits in the while loops in the algorithm are controlled by the values of the variables w and s, in the loops implementing these waits value false of these variables is restored, if necessary (see statements p3, p6.4 and p9). In Figure 2, the notation:

var*=*val;

is a shorthand for a resetting of the variable var to the value val, i.e. it is equivalent to the following conditional statement:

if (var!=val) var=val;

The third algorithm, presented in Figure 4, extends the first algorithm bit differently than the robust algorithm did. Namely, it enforces first-come first-served fairness property but for the price of an additional one-bit communication variable p (parity of the mutual exclusion request).

In the third algorithm, there is a gate section (statement g1-4) in the prologue that just takes a snapshot of each process status and registers it in two local bit vectors: la (its j-th bit shows whether the j-th process passed its gate section at the time of a snapshot) and lp (it stores each process' parity). In addition, two new states (sixth and seventh) are defined for each process as:

6) after-gate - two variables: a and w are set to true (aws=true,true,false). A process in the after-gate state has executed its gate section, but did not get into the waiting room yet.

7) advanced transient state - two variables: a and s are set to true (aws=true,false,true). A process in the advanced transient state is waiting until processes that executed gate section before it did access the critical section.

The only transition from the entry state leads now to the after-gate state and is made unconditionally. The distinction between the after-gate state and the entry state is important only in the gate section, where a process in the after-gate section has la bit set to true while a process in the entry state causes this bit to be set to false. In other parts of the algorithm, the new after-gate state and the entry state are equivalent.

```
communication variables::  a, w, s, p: boolean = false
private variables::            j: 0..n, k: 0..n = 0
                          la, lp: array[0..n-1] of boolean

p1:  ai=true;
g1:  for(j=0;j<n;j++) {
g2:     la[j]=wj; lp[j]=pj;
     }
g3:  pi=!pi;
g4:  wi=true;
p2:  for(j=0;j<n;j++) while(sj);
p3:  ai=false;
p4:  while(!si) {
p5:      for (j=0;j<n & !aj;j++);
p6:      if (j==n) { si=true;
p6.1:        for (j=0;j<n & !aj;j++);
p6.2:        if (j<n) si=false;
p6.3:        else { wi=false;
p6.4g:           for(j=0;j<n;j++) while(wj & !aj);
           }
         }
p7:      if (j<n) for (j=0;j<n & (wj | !sj);j++);
p8:      if (j!=i & j<n) {
p8.1:        si=true; wi=false;
         }
     }
d1:   while(k<n) {
g5:     for(k=0;k<n & (!la[k] | pk!=lp[k] | !wk & !sk);k++);
d2:     if (k<n) {
d2.1:       for(j=0;j<n;j++) if (j==i) ai=true;
d2.2:       else while(!aj & sj);
d2.3:       for(j=0;j<n;j++) if (j==i) ai=false;
d2.4:       else while(aj & sj);
        }
     }
p9:  for(j=0;j<i;j++) while(!aj & (wj |sj));
     Critical Section
e1:  si=false;
```

Figure 4. Four-Bit First-Come First-Served Algorithm

Unlike in the previous algorithm, a process in the exit state is eligible to access the critical section only if all processes that beat it at the gate already executed the critical section (see statements g5-d2). As previously, the critical section is accessed by the eligible processes in the order of their numbering. Processes that reached the exit state but are not eligible yet to access the critical section move to the advanced transient state. This transition is made in the order of numbering of processes that are not eligible yet to access the critical section. In the advance transient state, each process waits for others to leave the exit state, and then moves back into the exit state. This transition is also made in the order of processes' numbering (see statements d2.1-2.4). It should be noted that processes reach the advanced transient state only after a separation took place, therefore no process can be executing a leader selection code (statements p5-p6.4) at that time. Thus, the condition for leader selection can be simply a negation of the variable a (regardless of the value of the variable s).

The justification for an additional communication variable p is simple. If a process takes a snapshot of the other process in the gate state, then this other process can execute its critical section and return to the gate state before the first process gets to the wait loop in statement g5. Consequently, the process taking a snapshot needs to be able to recognize whether a process in the gate state did or did not execute the critical section while the snapshoting process was progressing from the gate section to the loop of statement g5.

Suppose that a process P1 finished its gate section before the other process P2 started it, and both processes did not access the critical section after the most recent execution of the gate section. Hence, the snapshot entry la[p1] in P2 is true and lp[p1] is equal to the value of the variable p in P1. Thanks to the loop in statement g5, the process P2 cannot reach the critical section before the process P1. If the process P2 is inside the waiting room in the subsequent separation, then the leader of this separation had to reach the transient state after the process P2 decided to move to the inside state, that, in turn, had to happen after the process P1 left the passive state. Consequently, also process P1 have to be inside the waiting room in the subsequent separation, so there is no deadlock on the loop in statement g5 in the discussed case.

If the process P1 already executed the critical section, then two new cases have to be considered. In the first case, a snapshot lp[p1] stores the value of parity of the process P1 associated with the original gate section execution. In this case, the process P1 will not cause a delay of the process P2 in the loop in statement g5. The second case happens when the variable lp[p1] in the process P2 stores the parity of P1 from the subsequent gate execution. This means that the snapshot of P2 in P1 would not delay the process P1 in its progress towards the critical section. The process P2 will be delayed by P1, but the process P2 cannot get into the waiting room without P1 being there in the same separation, so no deadlock is possible either. Finally, when gate executions of two processes intersect, then these processes will set the corresponding la bits to false before reaching

the loop in statement g5 and therefore none will be delayed by the other there. In summary, the loop in statement g5 together with the gate section g1-g4 enforces first-come first-served order of accessing the critical section without introducing any deadlocks.

```
communication variables::  a, w, s, p: boolean = false
private variables::            j, k: 0..n, c: 0..n-1 = n-1
                          la, lp: array[0..n-1] of boolean

p1:  ai=true;
g1:  for(j=0;j<n;j++) {
g2:     la[j]=wj; lp[j]=pj;
     }
g3:  pi=!pi;
g4:  wi=true;
p2:  for(k=1;k<n;k++) for(j=0;j<n;j++)
p3:      while(sj && i<>j) { ai*=*true; si*=*false; }
p4:  while(!si) { wi*=*true; ai*=*false;
p5:      for (j=0;j<n & !aj;j++);
p6:      if (j==n) { si=true;
p6.1:        for (j=0;j<n & !aj;j++);
p6.2:        if (j<n) si=false;
p6.3a:       else wi=false;
         }
p7:      if (j<n) for (j=0;j<n & (wj | !sj);j++);
p8:      if (j!=i & j<n) { si=true;
p8.1a:       if (!sj) si=false;
p8.1b:       else wi=false;
         }
     }
p6.4:for(j=0;j<n;j++) while(wj & !aj  && i<>j)
p6.4a:   { ai*=*false; wi*=*false; }
d1:  do {
g5:     for(k=0;k<n & (!la[k] | pk!=lp[k] | !sk);k++);
d2:     if (k<n) { c--;
d2.1:       for(j=0;j<n;j++) while(!aj & sj)
d2.2:           { ai*=*j>=i; wi*=*false; }
d2.3:       for(j=0;j<n;j++) while(aj & sj)
d2.4:           { ai*=*j<i; wi*=*false; }
        }
d3:  } while(k<n & c>0)
p9:  for(j=0;j<i;j++) while(!aj & sj  && i<>j)
p9a:    { ai*=*false; wi*=*false; }
     Critical Section
e1:  si=false;
```

Figure 5. Four-Bit Robust First-Come First-Served Algorithm

The last algorithm presented in Figure 5 provides first-come first-served robust mutual exclusion using just four one-bit communication variables. It is created by modifying and combining the robust linear wait algorithm in Figure 2 with the first-come first-served algorithm presented in Figure 4. It should be noted that, unlike the other communication variables, the variable p should not be reset to initial value at the end of abortions. In the first-come first-served algorithm, the code after the gate section is almost the same as in the basic algorithm shown in Figure 1. In the last algorithm presented in Figure 5, the after-gate code is nearly identical with the robust linear wait algorithm. The important difference between those two algorithms is that processes waiting for their turn to access critical section in the third algorithm can get deadlocked in the presence of transient malfunctions. As the result of transient malfunctions two or more processes may be placed immediately after the gate section with such values of their snapshot vectors la and lp that they will wait for each other in the loop of statement g5. In the robust algorithm in Figure 5 the separation of processes enables each process to discover such a deadlock. Statements d1-d3 at the end of prologue in Figure 5 are used to synchronize deadlock checking by all processes waiting after the gate.

If there is no deadlock, each cycle around the exit state and the advanced transient state makes at least one process in the exit state eligible to access the critical section. Thus, without a deadlock, each process can cycle around those two states no more then n-2 times. The counter c keeps track of the number of made cycles and is used in detecting and resolving the deadlock in statement d3.

It should be noted that all unbounded waits in conditions of while loops contain references to the variables a, w, s and to the negation of variable a. Consequently, all values of the variable a, and false values of the variables w and s are restored in the while loops, if necessary.

Due to the space limitation the more rigorous proofs of the presented algorithms' properties are omitted here.

4. Conclusion

The robust, fair mutual exclusion algorithms that are immune to several types of malfunctions were presented. These algorithms use fewer communication variables per process than any published algorithms with similar properties. The four-bit first-come first-served robust mutual exclusion algorithm contains just a single robust mutual exclusion algorithm as a basic component.

References

[1] Davidson, C.M., "A Note on Concurrent Programming Control," IEEE Transaction on Software Engineering, vol. SE-13, no. 7, July, 1987, pp. 865-866.

[2] Dijkstra, E.W. "Solution to a problem in concurrent programming control," Communication of the ACM, vol. 8, no. 9, September, 1967, p. 569.

[3] Eisenberg, M.A., and McGuire, M.R. "Further comments on Dijkstra's concurrent programming control problem," Communication of the ACM, vol. 15, no. 11, November, 1972, pp. 999.

[4] Ferguson, M.J. "Multiaccess in a Nonqueueing Mailbox Environment," IEEE Transaction on Software Engineering, vol. SE-10, no. 3, May, 1984, pp. 237-243.

[5] Knuth D.E., "Additional comments on a problem in concurrent programming control," Communication of the ACM, vol. 9, no. 5, May, 1966, p. 321-322.

[6] Lamport, L. "The mutual Exclusion Problem: Part I - A Theory of Interprocess Communication," JACM, vol. 33, no. 2, April, 1986, pp. 313-326.

[7] Lamport, L. "The mutual Exclusion Problem: Part II - Statement and Solutions," JACM, vol. 33, no. 2, April, 1986, pp. 327-348.

[8] Lycklama, E.A. "A First-Come First-Served Solution to the Critical Section Problem Using Five Bits," M.Sc. thesis, University of Toronto, October 1987.

[9] Lycklama, E.A. and Hadzilacos, V. "A first come first served mutual exclusion algorithm with small communication variables," submitted for publication, draft dated May 12, 1989.

[10] Morris, J.M. "A starvation-free solution to the mutual exclusion problem," Information Processing Letter, vol. 8, no. 2, 1979, pp. 76-80.

[11] Peterson, G.L. "A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables," ACM Transactions on Programming Languages and Systems, vol. 5, no. 1, January 1983, pp. 56-65.

[12] Raynal, M. "Algorithms for Mutual Exclusion," The MIT Press, Cambridge, Massachusetts, 1986.

[13] Szymanski, B.K. "A Simple Solution to Lamport's Concurrent Programming Problem with Linear Wait," Proc. 1988 International Conference on Supercomputing, St. Malo, France, July 4-8, 1988, pp. 621-626.

[14] Szymanski, B.K. "Efficient First-Come-First-Serve Mutual Exclusion Algorithm," Technical Report, RPI, Troy, NY, December, 1988.

[15] Truuvert K. "A Self-Stabilizing First-Come-First-Served Mutual Exclusion Algorithm With Small Shared Variables," Technical Note, University of Toronto, July, 1989.