SELF-STABILIZING ℓ-EXCLUSION:
A CORRECTNESS PROOF

by

MILOSLAV BESTA

PROSPECTUS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2002

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor Date

_____

_____

_____

Contents

List of Figures

Chapter 1

Introduction

Presently computer software is used in virtually every area of human activity. Software can be found not only on desktop computers, but also in many machines and devices that are used in manufacturing, communications, defense, transportation, aerospace, *e*-commerce, and health care. Software is used more and more to solve larger and more demanding problems. As the size of problems grows, software also becomes larger and more complex. At the same time, requirements on software performance, real-time behavior, fault tolerance, security, and dependability are increasing. Distributed computing has become an important part of the battle against larger and more demanding requirements.

In many areas, high requirements are imposed on fault tolerance of the computer system. A fault tolerant system must be reliable and must be able to tolerate failures up to some specified degree. Such systems are generally built from components that are fault tolerant as well. One of the vital parts of every distributed fault tolerant system is its ability to reliably share data and resources. For this task the use of a reliable, fault tolerant mutual exclusion algorithm (when several processes compete for a single resource) or ℓ-exclusion algorithm (when at most ℓ processes can use an instance of the resource at the same time) is essential.

Self-stabilizing algorithms are inherently fault tolerant. When a transient failure occurs in a system, the fault can affect shared data and put the system into an illegal

(unstable) state. When a self-stabilizing algorithm is used, eventually the system will stabilize into a legal state without any outside intervention or system re-initialization. Thereafter, the system behaves as if no transient fault ever occurred.

A self-stabilizing ℓ-exclusion algorithm combines an ℓ-exclusion algorithm with the self-stabilization after a transient failure occurs. The first and only self-stabilizing ℓ-exclusion algorithm that satisfies all the requirements imposed on self-stabilization and ℓ-exclusion was published by Abraham, Dolev, Herman, and Koll (1997, 2001). The algorithm there does not impose any restrictions on the communication topology and tolerates up to ℓ crashes of processes.

The above mentioned self-stabilizing ℓ-exclusion algorithm is complicated. Since the self-stabilizing ℓ-exclusion algorithm is intended to be used in fault tolerant systems, high requirements on correctness and reliability of the algorithm are imposed. To make sure the algorithm in (2001) is truly reliable and correct a rigorous correctness proof of the algorithm is needed.

This dissertation prospectus proposes to conduct research on correctness of the above mentioned algorithm. We propose to thoroughly analyze the algorithm and develop a correctness proof of the algorithm. As the methodology for the proof development, we propose to use the rigorous approach and notation of Linear Time Temporal Logic, see Manna and Pnueli (1992).

The remainder of the prospectus is organized as follows. In Chapter 2, we discuss the problem background and the concepts needed for presentation of the algorithm. In Chapter 3, the self-stabilizing ℓ-exclusion algorithm is presented. There, we discuss the structure of the algorithm, the subcomponents, and the underlying ideas of its implementation. In Chapter 4 the proposed areas of research and expected results are discussed. There, we also give an overview and justification of the methodology that is planned to be used.

Chapter 2

Problem Background

The problem of mutual exclusion is one of the fundamental problems in computer science. Assume that there exist N sequential processes sharing a resource. When a process executes part of its code known as the critical section, it is required that none of the other processes is in its critical section. Any process wishing to enter its critical section must do so within a finite amount of time. If a process is stopped outside its critical section, it cannot block the other processes from entering their critical sections. Nothing about the relative speeds of the processes is assumed. The first solution of the mutual exclusion problem was published by Dijkstra (1965), who generalized Dekker's solution of mutual exclusion between two processes. Later, several other solutions were given, e.g., by Peterson (1981), Lamport (1986b), and Lycklama and Hadzilacos (1991).

In 1973, Dijkstra introduced the concept of self-stabilization. The concept was demonstrated on the problem of mutual exclusion. The mutual exclusion problem can be described by an invariant on states. The invariant expresses that, in every state, there is at most one process in the critical section. That invariant holds in the initial state of the system as well as in all states that can be reached during execution of the system. When the shared memory is truly distributed, a local copy of the shared data is maintained. In case of a transient fault within a participating process, the shared data can become inconsistent and the system can reach an illegal state.

Consequently, the invariant of mutual exclusion is violated. Once an illegal state is reached, the system usually stays in an illegal state forever.

A system is called self-stabilizing if, regardless of the initial state and regardless of the transitions taken by the system, the system always reaches a legal state after taking a finite number of transitions and, thereafter, remains in a legal state. It is assumed that the system is fair and will take an infinitely enabled transition infinitely often. Thus, a self-stabilizing system recovers from a transient failure without outside intervention. Three self-stabilizing solutions of the mutual exclusion problem were given by Dijkstra in (1974).

Fischer, Lynch, Burns, and Borodin (1979) generalized the problem of mutual exclusion to the case where some number $\ell \geq 1$ of processes (but not more) are permitted to be simultaneously in their critical sections. If there are fewer than ℓ processes in their critical sections, it is possible for another process to enter its critical section. This mutual exclusion problem is called ℓ-exclusion.

There is a simple solution of the ℓ-exclusion problem that employs a mutual exclusion algorithm. This solution uses a queue of processes waiting to acquire one of the ℓ available instances of a resource. Once there is an instance of the resource available, the process at the head of the queue is allowed to acquire that resource. Access to the queue is synchronized by the mutual exclusion algorithm. Although this solution correctly implements the ℓ-exclusion problem, use of that solution leads to degradation of performance in cases when several resources are available for immediate assignment to processes that are already waiting to enter their critical sections. If the process waiting at the head of the queue is slow or delays its execution, the next waiting process is prohibited from leaving the queue, entering its critical section, and acquiring another available instance of the resource. Such a solution lacks concurrency, which is a limitation, particularly in distributed fault tolerant systems. In (1979), Fischer, Lynch, Burns, and Borodin present several solutions of the ℓ-exclusion problem with

different degrees of concurrency and robustness properties.

In 1986, Lamport introduced the notions of safe, regular, and atomic registers, see Lamport (1986c, 1986d). In his seminal work, Lamport analyzed the means of communication in truly distributed systems. When a shared register is implemented in hardware, assumptions about the relative speeds of the communicating processes are made, and delays are introduced into the system to synchronize accesses to a shared register. At a higher level of data sharing, particularly in distributed systems, when the communicating processes are running at vastly different speeds, the wait and delay approach can lead to a substantial degradation of concurrency and performance of the system. When no assumptions about the relative speeds of the communicating processes are made, a shared register implementation cannot rely upon any delays. Then, three classes of register implementation are possible, see Lamport (1986d).

The weakest possibility is a safe register, for which it is assumed only that a read operation that is not concurrent with any write operation returns the correct, i.e., the most recently written value. If there is a concurrent write operation, the read operation returns any of the admissible values for the register.

The next stronger possibility is a regular register. If a read operation is not concurrent with any write operation, the most recently written value is returned as in the case of a safe register. If a read operation is concurrent with one or more write operations, then the read operation returns either the old value, i.e., the value that had been most recently written to the register before the read operation started, or one of the new values that have been written to the register concurrently with the read operation. As a consequence, when a write operation is concurrent with two read operations, it is possible that the first read operation returns the new value, but the second read operation returns the old value.

The last possibility is an atomic register. It is a safe register in which all read and write operations behave as if there were no concurrent operations in the system.

In other words, it is possible to find some sequential ordering of operations such that the results of the operations executed concurrently will be the same as the results of the operations executed in that sequential order. Consequently, once a read returns a new value, all subsequent reads operations return either the new value or even a newer value.

There is another criterion according to which one can categorize shared registers. A register can be read by a single process only or by multiple processes. Similarly, a register can be written by one or multiple processes. To express this fact we describe registers as single/multiple-reader and single/multiple writer registers. For example, a single-writer multiple-reader register is a register that can be written by only one process, but multiple processes can read the contents of that register.

Chapter 3

Self-Stabilizing ℓ-Exclusion

There have been very few attempts to design a self-stabilizing solution to the mutual exclusion problem. The first self-stabilizing mutual exclusion algorithm was given by Dijkstra (1974). His solution uses the ring-shaped topology of process communication, the shared memory model, and a special node running a different version of the algorithm than the remaining nodes. Although his solution is of limited practical use, it showed the existence of self-stabilizing algorithms and their usefulness in fault tolerant systems.

A major contribution to solving the problem of self-stabilizing mutual exclusion was made by Lamport (1986b). In his work, he defined several properties and requirements that one might impose on a mutual exclusion algorithm and presented three self-stabilizing solutions of the mutual exclusion problem, each of them satisfying a stronger set of the requirements. For example, his one-bit solution satisfies the basic mutual exclusion and deadlock freedom properties. The three-bit solution satisfies all the properties of the one-bit solution plus it satisfies the strong-fairness property, i.e., a process trying to enter the critical section infinitely often will be in the critical section infinitely often. If a system consists of N concurrent processes competing for a resource, the N-bit solution given by Lamport (1986b) satisfies all the requirements imposed on the three-bit solution and also the first-come, first-served requirement on the order of processes that try to enter the critical section.

The self-stabilizing solution of the mutual exclusion problem of Dijkstra (1974) was generalized by Flatebo, Datta, and Schoone (1994). In that paper, the authors implement the self-stabilizing ℓ-exclusion problem using the token-ring communication topology. In their implementation, all processes including those that are not interested in entering the critical section have to participate in passing the token in the ring. Consequently, the system is not able to self-stabilize in case of a permanent failure of one of the processes. This is a drawback of their solution. Also the prescribed ring communication topology is of limited use.

The first fully satisfiable self-stabilizing ℓ-exclusion algorithm was published by Abraham, Dolev, Herman, and Koll (2001). (A preliminary version of the paper was published in 1997). The algorithm there satisfies all the ℓ-exclusion and self-stabilization requirements, tolerates crashes of processes, and uses a set of single-writer multiple-reader regular registers. A detailed description of the problem follows.

Assume there exist N processes that share $\ell < N$ instances of a resource. When a process intends to use an instance of the resource, it executes a so-called trying section in order to obtain permission to use the resource. After obtaining the permission, it executes its critical section, and thereafter a so-called exit section, which is an obligation to release the resource. Thereafter, the process executes its remainder section where it does not use the shared resource. Thus, the structure of a program that executes the ℓ-exclusion algorithm can be described as follows.

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

A solution to the ℓ-exclusion problem is an algorithm for the trying and exit sections that satisfies the ℓ-exclusion requirements. A self-stabilizing ℓ-exclusion algorithm can start in any arbitrary state and after a finite amount of time the ℓ-exclusion property holds and thereafter remains to hold. Thus, a self-stabilizing algorithm can recover from transient faults that leave the system in an illegal state. The algorithm by Abraham, Dolev, Herman, and Koll (2001) also tolerates up to ℓ crashes of processes.

A process crashes when it stops executing its code. If a crash occurs, the local variables and shared registers of that process are assumed to be set to arbitrary values. Thereafter, the process does not modify the contents of its variables and registers. From the point of view of another process, it is undistinguishable whether a process is crashed in its remainder section or it is executing its remainder section forever. The same holds for a process crashed in or forever executing its critical section. Whether a process is in its critical or remainder section is determined by the values recorded in shared registers.

A process exhibits a transient failure when it sets its local variables or shared registers or both to some arbitrary values, but, thereafter, it continues executing its code. Because of the random modification of values in the local variables and shared registers, the failed process as well as the other processes in the system can be in an illegal state and behave incorrectly. Yet, because of self-stabilization, the system eventually reaches a legitimate state and thereafter behaves correctly.

In our correctness proof, we may assume the system starts in an arbitrary state. This reflects the state of the system after the last crash or transient fault. We have to establish that the properties of a self-stabilizing ℓ-exclusion hold. Those properties have been formulated by Abraham, Dolev, Herman, and Koll (2001) as follows.

Requirement 1 (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 (Liveness): *In an execution in which less than ℓ processes are crashed, each non-crashed 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.*

Definition 1 (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.*

The algorithm by Abraham, Dolev, Herman, and Koll is designed as a combination of two main components: the first one implements the safety requirement, and the second one implements the liveness requirement. In general, the algorithm implements a mechanism of counting processes that may be in their critical section and introduces a system of rotating process identities among processes that compete for entering the critical section in order to maintain liveness.

The liveness requirement is ensured by the so-called isolation algorithm. The goal of that algorithm is to dynamically order processes in a way that ensures that none of the processes that are trying to enter the critical section will starve. For that purpose, the algorithm uses N shared single-writer multiple-reader regular registers, each of size ℓ. These registers may also be viewed as an $N \times \ell$ matrix ORD of binary variables. Furthermore, an array X of shared boolean registers is employed to keep track of processes that are interested in entering their critical section. Each process P_i writes row i of matrix ORD and modifies element X_i of the array X. If element X_i is set to *false*, process P_i is not interested in entering its critical section. Therefore, row i of matrix ORD is eliminated from the process of computing the dynamic identity.

The dynamic identities are computed as follows: After the rows corresponding to the processes that are not interested in entering their critical sections are eliminated, the first column of the matrix is used to compute the identity of the process that can

enter the critical section with the highest priority. If the value of the first element in that column is the same as the value in the last element of the column, the index of that first element identifies the highest priority process. Otherwise, the algorithm finds the minimal index in that column such that the corresponding element contains a value that differs from the value in the first element. That index identifies the highest priority process. Subsequently, the row of the process that has been selected above is eliminated from matrix *ORD* and the same algorithm is applied on the second column to determine the process with the next highest priority. The algorithm stops when all ℓ columns are used to determine process priorities or when there are no more processes competing for a shared resource.

Once a process P_i leaves its critical section, it modifies its row in matrix *ORD*, thereby enabling other processes to enter their critical sections before process P_i will be granted access to its critical section again. If process P_i is crashed, it does not modify its row in matrix *ORD*. Because non-crashed processes rotate the values in columns of matrix *ORD*, eventually the isolation algorithm will assign a highest priority to process P_i. If less than ℓ processes are crashed in the system, the non-crashed processes will compete for those resources that are not consumed by the crashed processes.

The safety requirement is implemented by the so-called safe algorithm. The main goal of this component is to count the processes that may be in their critical sections and, if the count exceeds ℓ, to disallow the process to enter. To count the processes that are in the critical section the algorithm employs a shared array *TRY* of regular boolean registers. When an element with index i (denoted TRY_i) is set to *true*, process P_i may be in its critical section. When TRY_i is *false*, P_i is outside of its critical section.

When a slow running process P_i reads the contents of array *TRY*, the read values may indicate that a fast running process P_j is continuously in its critical section. In

begin

$\mathbf{d}_1$ *read-all-VEC*(v);

$\mathbf{d}_2$ *vector*:= *report*(v);

$\mathbf{d}_3$ **write**$(VEC_i := vector)$;

$\mathbf{d}_4$ *read-all-VEC*(v);

end

Figure 3.1: Diagonal algorithm

reality, process P_j repeatedly enters and leaves its critical section. Because process P_i may always read the register TRY_j at the moment when P_j is in its critical section, it may wait forever for the permission to enter.

To ensure that a slow process will eventually enter the critical section, the safe algorithm employs the so-called diagonal algorithm. The diagonal algorithm keeps track of fast processes. Fast processes are not counted by slow processes as being in the critical section. A slow process assumes that a fast process will correctly determine which processes are in the critical section. If ℓ processes are already in the critical section, the fast process will not enter its critical section. In the following paragraphs, we explain how the diagonal algorithm works.

As before, assume there are N processes $P_1, \ldots, P_N$ in the system. Each process is identified by its index i. We define a *color* as an integer in $[0, \ldots, 2N-1]$. To determine if a process runs fast, the diagonal algorithm uses the assumption that a fast process frequently modifies its shared registers. For that purpose, the diagonal algorithm uses a set of N regular vectors. This set of vectors is represented by matrix VEC with N rows. Row i of the matrix is a regular vector denoted by VEC_i.

Let i be an index of process P_i and v_i be the row with index i of matrix VEC. For indices $j \neq i$, we define element $v_i[j]$ to be a pair of colors denoted $v_i[j].first$ and $v_i[j].second$, and the element $v_i[i]$ to be a single value denoted $v_i[i].color$. Hereafter,

function *report*(v) *(by process* P_i*)*

begin

select $d \in \{color |\ 0 \leq color \leq 2N-1 \wedge color \neq v_i[i].color$

$\wedge\ color \notin \bigcup_{i \neq j}\{v_j[i].first,\ v_j[i].second\}\}$;

$vector[i] := d$;

forall $1 \leq j \leq N \wedge j \neq i$ **concurrently do**

$vector[j].second := vector[j].first$;

$vector[j].first := v_j[j].color$;

return *vector*;

end

Figure 3.2: Implementation of function *report*

the value of $v_i[i].color$ is called the color of process P_i.

Let P_i and P_j, $i \neq j$, be two processes executing the diagonal algorithm and v_i and v_j be the vectors containing the values of rows VEC_i and VEC_j, respectively. As defined by Abraham, Dolev, Herman, and Koll in (2001), process P_i dominates P_j if $v_i[j].first = v_i[j].second = v_j[j].color$. For example, if $N = 4$, $v_2 = ((3,8),3,(7,4),(1,1))$, and $v_4 = ((2,2),(5,5),(6,1),1)$, then process P_2 dominates P_4, but process P_4 does not dominate P_2.

The basic idea behind the diagonal algorithm in Figure 3.1 is the following. As the process P_i runs, it repeatedly executes the statements at locations $\mathbf{d}_1$, $\mathbf{d}_2$,$\mathbf{d}_3$, and $\mathbf{d}_4$. When the statement at location $\mathbf{d}_2$ is executed, a new, unused color of process P_i is selected and recorded in $vector[i].color$. At the same time, current and previously current colors of every other process P_j are recorded in $vector[j].first$ and $vector[j].second$. At the moment *vector* is written to VEC_i, no other process can dominate P_i because the equality from the definition of process domination cannot hold. On the other hand, process P_i may dominate a process P_j if process P_j did not

```
procedure read-all-VEC(v)
begin
    forall 1≤j≤N concurrently do
        read(v_j := VEC_j);
end
```

Figure 3.3: Implementation of procedure *read-all-VEC*

write a new color recently. In that case, process P_i may determine that the current color of process P_j is the same as the recorded current and previously current colors.

The actual implementation of the diagonal algorithm (see Figure 3.1) uses two local variables. Matrix v is a local copy of the matrix *VEC*, and *vector* is an array used to compute the new value of the vector VEC_i. The algorithm uses two subroutines. Function *report*(v) (in Figure 3.2) is used to compute the new value of variable *vector*, and procedure *read-all-VEC*(v) (in Figure 3.3) is used to read the values of matrix *VEC* into local variable v. When process P_i executes the diagonal algorithm, it repeatedly reads the shared matrix *VEC* and records those values into v. Then it invokes the function *report*(v) to determine the value of *vector*. Thereafter, that value is written to VEC_i.

When function *report* is executed by process P_i (see Figure 3.2), all values recorded in column i of matrix v are collected. There are $2N-1$ elements in that column ($N-1$ pairs in rows different from i and a single element in row i). Function *report* selects a new color d that is not used in that column. Such a color exists, because there are $2N$ colors in total, but at most $2N-1$ occur in the column under consideration. Then the new color d is recorded in element $vector[i].color$. Later, when the statement at location $\mathbf{d}_3$ (see Figure 3.1) is executed, that color is recorded in $VEC_i[i].color$. The rest of the values of *vector* are determined as follows: For every index j, $j \neq i$, the algorithm shifts the color $vector[j].first$ into $vector[j].second$ and writes color

begin

$\mathbf{c}_1$ *try*:= *false*;

$\mathbf{c}_2$ **write**(TRY_i := *false*);

$\mathbf{c}_3$ **repeat**

$\mathbf{c}_{3.1}$ $\boxed{\textit{read-all-VEC}(v);}$

$\mathbf{c}_{3.2}$ $\boxed{\textit{vector}:= \textit{report}(v);}$

$\mathbf{c}_{3.3}$ *old-try*:= *try*;

$\mathbf{c}_{3.4}$ *try*:= *getTry*();

$\mathbf{c}_{3.5}$ $\boxed{\textbf{write}(TRY_i := \textit{try});}$

$\mathbf{c}_{3.6}$ **if** *try* $\wedge \neg$*old-try* **then write**(VEC_i := *vector*);

$\mathbf{c}_4$ **until** *try*;

$\mathbf{c}_5$ **concurrently do**

begin $\boxed{\textit{read-all-VEC}(v);}$ *read-all-TRY*(t)) **end**

$\mathbf{c}_6$ $B := \{j \mid t_j \wedge \neg(v_i \text{ dominates } v_j)\}$

$\mathbf{c}_7$ **if** $|B| > \ell$ **then goto** $\mathbf{c}_3$;

$\mathbf{c}_8$ CRITICAL SECTION

$\mathbf{c}_9$ **write**(TRY_i := *false*);

$\mathbf{c}_{10}$ **write**(X_i := *false*);

$\mathbf{c}_{11}$ v:= *change_order*(M, i);

$\mathbf{c}_{12}$ **write**(ORD_i := v);

end

Figure 3.4: Combined algorithm

$v_j[j].color$ into $vector[j].first$. Thus, process P_i records in $vector[j].first$ the most recently read color of process P_j and in $vector[j].second$ the previously read color of process P_j.

In Figure 3.4, we give the combined self-stabilizing ℓ-exclusion algorithm. We use frames to mark the statements that are part of the diagonal algorithm in Figure 3.1. The combined algorithm is complicated. This prospectus gives only a high-level overview of the self-stabilizing ℓ-exclusion algorithm. A complete description of that algorithm will be given in the dissertation thesis. At this moment, we use the code of the combined algorithm to illustrate how the safe and liveness components are combined and how the diagonal algorithm is embedded in the safe algorithm.

The safe algorithm is formed by the statements at locations $\mathbf{c}_1$–$\mathbf{c}_{3.3}$ and $\mathbf{c}_{3.5}$–$\mathbf{c}_9$. The diagonal algorithm is a part of that safe component. Notice that the statement at location $\mathbf{d}_1$ of the diagonal algorithm (Figure 3.1) appears at location $\mathbf{c}_{3.1}$ of the safe algorithm. Similarly, location $\mathbf{d}_2$ maps to $\mathbf{c}_{3.2}$, location $\mathbf{d}_3$ maps to $\mathbf{c}_{3.5}$, and location $\mathbf{d}_4$ maps to $\mathbf{c}_5$. The correspondence between the statements of the safe and the diagonal algorithms and the fact that the variables used in the diagonal algorithm are not modified anywhere else in the code of the combined algorithm lead us to a conclusion that it will be possible to formulate and prove most of the properties about the diagonal algorithm independently of the safe and combined algorithms and reuse those properties later to reason about the safe and combined algorithms.

The isolation algorithm is implemented by the statements at locations $\mathbf{c}_{3.4}$, $\mathbf{c}_{10}$, $\mathbf{c}_{11}$, and $\mathbf{c}_{12}$. Function *getTry* (at location $\mathbf{c}_{3.4}$) implements the main part of the isolation algorithm and returns *true* only if the isolation algorithm determines that process P_i has high enough priority to enter the critical section. Procedure *change_order* (at location $\mathbf{c}_{11}$) implements the rotation of the process identities. We omit the implementation of both these functions from this prospectus, because they are complicated. The reader can find the code in Abraham, Dolev, Herman, and Koll (2001). A detailed

description of these functions will appear in the dissertation thesis.

Similar to the diagonal algorithm, the isolation algorithm is also embedded in the combined algorithm through some mapping. We envision that this will allow us to develop a correctness proof of the isolation algorithm independently and later carry the properties over on the combined algorithm.

An essential part of the combined algorithm can be found at locations $\mathbf{c}_6$ and $\mathbf{c}_7$ in Figure 3.4. At location $\mathbf{c}_6$ the algorithm counts all processes that are attempting to enter the critical section. Those processes have to receive permission from the isolation algorithm, i.e., function *getTry* must return *true*. The contents of variable *try* is subsequently written into the register TRY_i. A local copy of array TRY is kept in variable t and, at location $\mathbf{c}_6$, used to determine the number of processes that are attempting to enter the critical section. A process P_j is excluded from being counted by process P_i if it dominates to P_i. The reason for that is that, otherwise, slow running processes could starve, as it has been explained earlier in this chapter. At location $\mathbf{c}_7$, if the number of processes that could be in the critical section does not exceed ℓ, process P_i is allowed to enter the critical section. After the process leaves its critical section it resets its register TRY_i and rotates identities to allow other processes to enter their critical section.

Chapter 4

Proposed Research Goals and Methodology

As we saw in the previous chapter, the self-stabilizing ℓ-exclusion algorithm is complicated. The complexity of that algorithm as well as the importance of its correctness justify the need of a rigorous correctness proof of that algorithm. This is the main aim of the dissertation thesis.

In the first phase, we have focused our research on understanding and analyzing the algorithm, as it has been described by Abraham, Dolev, Herman, and Koll in (1997, 2001). We have already spent a substantial amount of time on the analysis of the components of that algorithm, and presently we understand all basic properties and behavior of the entire algorithm. For designing a correctness proof, however, many more properties of the algorithm will need to be discovered, formulated, and proved.

In the next phase, which is still a preparatory phase for correctness proof development, we have worked on the computational model of the system. The standard model employs states and histories. A state is a mapping from variables to values. A history records events in the system that are related to access to the shared registers. The program is represented by a guarded command language similar to UNITY. For a description of the UNITY notation see Chandy and Misra (1988). The program semantics is the usual semantics of a UNITY program. The only two operations for which the semantics has to be defined are the regular register `read` and `write` op-

erations. These operations are not atomic when used to access regular registers and their semantics has to reflect this fact. Therefore, the initialization and the termination of those operations have to be recorded in the history. At this moment, we are also considering to assume that the ℓ-exclusion algorithm employs atomic registers instead of regular registers. If we opt for this register representation, we will be able to simplify the computational model and semantics of the program. This would also simplify the correctness proof.

The core phase of our research will be focused on developing a correctness proof of the self-stabilizing ℓ-exclusion algorithm. We have decided to use Linear Time Temporal Logic, see Manna and Pnueli (1992, 1995), as the proof design methodology. Linear Time Temporal Logic allows one to express several temporal properties, such as the always ($\Box$) and the eventually ($\Diamond$) properties. In Linear Time Temporal Logic, one can also express, using the so-called weak-until ($\mathcal{W}$) operator, that a property holds until another property holds. In this logic one can reason about non-terminating executions of the system as well. Therefore, the logic is convenient to use for proof development of concurrent, non-terminating systems.

There are several other proof methodologies that are used for formal proofs development. The first one is the Hoare's verification method, which is based on Hoare logic of sequential processes, see Hoare (1969). In Hoare's method each program statement is augmented with two state predicates — a precondition and a postcondition. To show a property is satisfied when a program terminates one has to come up with strong enough pre- and postconditions for every statement in the program and show that, when the rules of Hoare logic are applied, the postcondition for the entire program holds. Because Hoare logic does not offer any means for parallel composition of processes, it is not directly applicable to concurrent programs. It also does not allow one to express non-trivial properties and invariants of programs that do not terminate.

Owicki and Gries (1976) generalized Hoare's verification method to concurrent systems, where communication takes place through shared variables. In their work, Owicki and Gries introduced the inference-freedom test to ensure that the assertions attached to sequential statements are not invalidated by executions of statements in other processes. Since this method is based, in essence, on the same principles as Hoare's verification method, Owicki-Gries method is not suitable for expressing invariant properties of non-terminating programs.

We plan to develop our correctness proof in several stages, which correspond to the design of the ℓ-exclusion algorithm. The task of a correctness proof development is known to be hard. Detailed understanding of the algorithm and its properties is required. Formulation and proofs of many auxiliary properties are necessary. During the proof development phase, one may find errors in the algorithm. Should this be the case, we will attempt to correct the algorithm and develop a correctness proof of the corrected algorithm.

While a correctness proof is being developed, one may notice that the algorithm could be improved in some way. For example, it may be possible to use less shared space to implement the algorithm, or a more efficient version of the code may be suggested. We will pay attention to this issue and if we discover some improvements, we will incorporate them into the algorithm and give a correctness proof of the improved version of the algorithm.

We will also pay a close attention to the methods and techniques that we use in our correctness proof. The algorithm is composed from several independent components that are nested one into another. We believe it is possible to develop a correctness proof for each component independently and later combine the proofs together to a correctness proof of the ℓ-exclusion algorithm. This is a non-trivial issue in proofs of concurrent algorithms. If the technique of proof composition is generalized, it can be reused in proofs of many other concurrent algorithms.

Chapter 5

Conclusion

In this dissertation prospectus we discussed the self-stabilizing ℓ-exclusion algorithm by Abraham, Dolev, Herman, and Koll (2001). In Chapter 2, we covered the problem background and some basic concepts employed in the algorithm. The ℓ-exclusion algorithm was presented in Chapter 3. The proposal and directions of our research are given in Chapter 4. We propose to develop a correctness proof of the self-stabilizing ℓ-exclusion algorithm. The correctness proof will allow one to conclude that the algorithm is correct and reliable. This will allow use of that algorithm for implementation of distributed self-stabilizing systems that employ ℓ-exclusion as the means of process synchronization.

Bibliography

Abadi, M. and L. Lamport (1991). The existence of refinement mappings. *Theoretical Computer Science 82*(2), 253–284.

Abraham, U. (1995). On interprocess communication and the implementation of multi-writer atomic registers. *Theoretical Computer Science 149*(2), 257–298.

Abraham, U., S. Dolev, T. Herman, and I. Koll (1997). Self-stabilizing ℓ-exclusion. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pp. 48–63. Carleton University Press.

Abraham, U., S. Dolev, T. Herman, and I. Koll (2001). Self-stabilizing ℓ-exclusion. *Theoretical Computer Science 266*(1–2), 653–692.

Afek, Y., D. Dolev, E. Gafni, M. Merritt, and N. Shavit (1994). A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16*(3), 939–953.

Beauquier, J. and S. Kekkonen-Moneta (1997). Fault-tolerance and self-stabilization: Impossibility results and solutions using self-stabilizing failure detectors. *International Journal of Systems Science 28*(11), 1177–1187.

Burns, J. E., M. G. Gouda, and R. E. Miller (1993). Stabilization and pseudo-stabilization. *Distributed Computing 7*, 35–42.

Chandy, K. M. and J. Misra (1988). *Parallel Program Design: A Foundation*. Addison-Wesley.

Cypher, R. (1995). The communication requirements of mutual exclusion. In *Pro-*

ceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'95, Santa Barbara, California, pp. 147–156.

de Roever, W.-P., F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers (2001). *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press.

Debest, X. A. (1995). Remark about self-stabilizing systems. *Communications of the ACM 38*(2), 115–117.

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

Dijkstra, E. W. (1974). Self-stabilizing systems in spite of distributed control. *Communications of the ACM 17*(11), 643–644.

Dolev, S. (2000). *Self-Stabilization*. MIT Press.

Dolev, S. and T. Herman (2001). Dijkstra's self-stabilizing algorithm in unsupportive environments. *Lecture Notes in Computer Science 2194*, 67–81.

Dolev, S., A. Israeli, and S. Moran (1993). Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing 7*, 3–16.

Fischer, M. J., N. A. Lynch, J. E. Burns, and A. Borodin (1979). Resource allocation with immunity to limited process failure. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pp. 234–254. IEEE.

Fischer, M. J., N. A. Lynch, J. E. Burns, and A. Borodin (1989). Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems (TOPLAS) 11*(1), 90–114.

Flatebo, M., A. K. Datta, and A. A. Schoone (1994). Self-stabilizing multi-token rings. *Distributed Computing 8*(3), 133–142.

Gouda, M. G. (1995). *The Triumph and Tribulation of System Stabilization*, Volume 972 of *Lecture Notes in Computer Science*, pp. 1–18. Springer-Verlag.

Hailpern, B. and S. S. Owicki (1982). Modular verification of concurrent programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pp. 322–336. ACM Press.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM 12*(10), 576–580.

Koymans, R., J. Vytopil, and W. P. de Roever (1983). Real-time programming and asynchronous message passing. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pp. 187–197.

Lamport, L. (1974). A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM 17*(8), 453–455.

Lamport, L. (1980). The 'Hoare logic' of concurrent programs. *Acta Informatica 14*, 21–37.

Lamport, L. (1983). Reasoning about nonatomic operations. In *Proceedings of the 10th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 28–37. ACM Press.

Lamport, L. (1984). 1983 invited address: Solved problems, unsolved problems and non-problems in concurrency. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pp. 1–11.

Lamport, L. (1986a). The mutual exclusion problem: Part I – A theory of interprocess communication. *Journal of the ACM (JACM) 33*(2), 313–326.

Lamport, L. (1986b). The mutual exclusion problem: Part II – Statement and solutions. *Journal of the ACM (JACM) 33*(2), 327–348.

Lamport, L. (1986c). On interprocess communication, part I: Basic formalism.

Distributed Computing 1(2), 77–85.

Lamport, L. (1986d). On interprocess communication, part II: Algorithms. *Distributed Computing 1*(2), 86–101.

Lamport, L. (1987). A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS) 5*(1), 1–11.

Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*(3), 872–923.

Lamport, L. and N. Lynch (1990). *Distributed Computing: Models and Methods*, pp. 1157–1199. MIT Press.

Lycklama, E. A. and V. Hadzilacos (1991). A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13*(4), 558–576.

Lynch, N. A. (1997). *Distributed Algorithms (Data Management Series)*. Morgan Kaufmann Publishers.

Manna, Z. and A. Pnueli (1983). How to cook a temporal proof system for your pet language. In *Proceedings of the 10th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 141–154. ACM Press.

Manna, Z. and A. Pnueli (1992). *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag.

Manna, Z. and A. Pnueli (1995). *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag.

Owicki, S. S. and D. Gries (1976). An axiomatic proof technique for parallel programs. *Acta Informatica 6*, 319–340.

Peterson, G. L. (1981). Myths about the mutual exclusion problem. *Information Processing Letters 12*(3), 115–116.

Pnueli, A. (1981). The temporal semantics of concurrent programs. *Theoretical Computer Science 13*(1), 45–60.