SELF-STABILIZING ℓ-EXCLUSION: A CORRECTNESS PROOF

by

MILOSLAV BESTA

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2005

MAJOR: COMPUTER SCIENCE

Approved by:

Advisor Date

DEDICATION

To my parents.

ACKNOWLEDGEMENTS

I wish to express my deepest appreciation to my advisor, Dr. Frank Stomp. It was his guidance, encouragement, enthusiasm and support that helped me to become a researcher. Without him, I would have never learned so much.

My parent's constant love and support were present in everything I attempted. I will never be able to pay them back for what I have received, and I will be forever grateful to them.

I would like to thank my friends for their support during the fun times as well as in the difficult times. My study at Wayne State University would not nearly be as enjoyable without them.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Formal Methods in Software Verification

Many software systems are required to satisfy the highest standards as to their reliability and dependability. The traditional approaches of software design, development, and testing are limited with respect to ensuring reliability and dependability. In applications where ultra-reliability is necessary, e.g., in communications, defense, transportation, aerospace, *e*-commerce, and health care, even the most thorough testing of software may not ensure a satisfactory level of reliability. In other applications, particularly in concurrent and distributed systems, thorough testing is almost impossible because of the nature of the concurrent environment. To tackle reliability and dependability in the above cases, a more rigorous approach, called formal methods, needs to be employed.

Formal methods is a field of computer science in which mathematical approaches are used for software development and verification. These approaches support the rigorous specification and verification of computer programs. Using formal methods in a program's specification means that the program's requirements are expressed using a formal language. To employ formal methods in a program's verification means that the program is represented by a system that has formal (mathematical) semantics. Notations and languages with defined mathematical meaning provide a means for the precise and unambiguous description of the program's behavior and the program's specification. This enables the use of rigorous mathematical techniques. Thus it is possible to devise a formal proof that a program satisfies its specification.

There are several reasons for using formal methods in software verification. The main reason is that a formal correctness proof of a program results in an extremely high level of confidence in the program's correctness. Moreover, devising a formal correctness proof may have several side effects. First, we can gain deep insight in the program and its behavior. This can result in discovering errors in the program or in strengthening the program's specification. In addition, that insight can lead to improvements of the program and to discoveries of novel techniques or methods reusable in other application contexts.

1.2 Problem Overview

Every computer program utilizes some elementary algorithms, which form the basic building blocks of the program. As an example, mutual exclusion is employed in nearly every distributed system, see, e.g., (Dijkstra 1965; Dijkstra 1974; Dolev 2000; Lamport 1974; Lamport 1986a; Lamport 1986b; Lamport 1987; Lamport and Lynch 1990; Lynch 1996). The problem of mutual exclusion was generalized to ℓ-exclusion in (Fischer, Lynch, Burns, and Borodin 1979; Fischer, Lynch, Burns, and Borodin 1989). According to (Fischer, Lynch, Burns, and Borodin 1979; Fischer, Lynch, Burns, and Borodin 1989), for $\ell \geq 1$, an ℓ-exclusion algorithm is to satisfy the (safety) property that at most ℓ processes can simultaneously execute their critical sections; and the (liveness) property that every process wishing to enter its critical section eventually enters the critical section; moreover, if less than ℓ processes execute their critical sections, additional processes may enter their critical sections. (The number of the additional processes is constrained by the safety property.)

Consider a distributed system that is required to be ultra reliable. If a transient fault occurs in the system, that fault can corrupt the local and/or the shared memory.

Thus the system's data may be arbitrarily modified, which can put the system into an illegal (unstable) state. If the system is self-stabilizing (Dolev, Israeli, and Moran 1993; Dolev 2000), it eventually stabilizes to a legal state with no outside intervention and, thereafter, it behaves as if no fault ever occurred. Thus, self-stabilizing systems are inherently fault-tolerant and dependable.

A self-stabilizing ℓ-exclusion (SLEX) algorithm is an extension of ℓ-exclusion, which is able to recover without outside intervention from transient faults and from a limited number of persistent faults. For instance, a SLEX algorithm can be employed in every distributed system that tolerates crashes of a limited number of processes and in which the shared data may be corrupted during a process crash.

The first solution to SLEX appears in (Abraham, Dolev, Herman, and Koll 2001). That solution uses the shared memory model, where the shared memory consists of single-writer multiple-reader regular registers (Lamport 1986d).

Some operational arguments supporting correctness of the above SLEX algorithm are presented in (Abraham, Dolev, Herman, and Koll 2001). Many of those arguments involve non-constructive reasoning. This means that many auxiliary quantities are not formulated explicitly and the validity of the properties is established by contradiction, i.e., by disproving their non-existence.

In this thesis, we present a constructive, assertional correctness proof of the complicated SLEX algorithm introduced in (Abraham, Dolev, Herman, and Koll 2001). Our proof is formulated in Linear–Time Temporal Logic (Manna and Pnueli 1992) and utilizes a history (Herlihy and Wing 1990) to model access to regular registers. That proof has led to some new insight in the algorithm. That has allowed us to identify some possible improvements of the SLEX algorithm in (Abraham, Dolev, Herman, and Koll 2001), as discussed below.

The possibility that processes may be crashed is considered in (Abraham, Dolev, Herman, and Koll 2001), which contributes to the complexity of the SLEX problem studied in this thesis. Intuitively, a process is crashed if it can never take any transition. A process that forever executes a code which does not modify any shared registers cannot be distinguished from a process that is crashed. Therefore, such a process is also considered to be crashed.

The part of code where the shared resource may be accessed is called the critical section. The SLEX algorithms in (Abraham, Dolev, Herman, and Koll 2001) and in this thesis satisfy the (safety) property that if at most ℓ processes are crashed in the critical section, from some point onwards at most ℓ processes can simultaneously be in their critical sections.

The SLEX algorithm in (Abraham, Dolev, Herman, and Koll 2001) satisfies the following (liveness) property: If less than ℓ processes are *crashed in the critical section or crashed in a state expressing that those processes are attempting to enter the critical section*, then every non-crashed process eventually enters the critical section. The insight we have gained during our analysis helped us to improve the SLEX in (Abraham, Dolev, Herman, and Koll 2001) so that if less than ℓ processes are *crashed in the critical section*, every non-crashed process eventually enters its critical section. Thus, the improved algorithm satisfies a stronger liveness property and tolerates crashes of processes outside the critical section that are not tolerated in (Abraham, Dolev, Herman, and Koll 2001).

The correctness proof presented in this thesis is on our improvement of the SLEX in (Abraham, Dolev, Herman, and Koll 2001). In our proof, we present a property characterizing processes (and their minimum number) identified by some process as attempting to enter their critical sections, see Lemma 7.2.21. To structure the liveness proof, we present a novel proof rule for reasoning about programs in the presence of disabled (crashed) processes. That proof rule, presented in Lemma 7.3.28, is similar to the proof rules in (Manna and Pnueli 1991) for reasoning about eventuality properties, in the sense that recursive applications of the proof rule are required.

The remainder of this thesis is organized as follows. In Chapter 2 we discuss the background of self-stabilizing ℓ-exclusion and some related work. Chapter 3 introduces some preliminary concepts used in the rest of the thesis. A description of our improved SLEX algorithm is presented in Chapter 4. The program semantics is described in Chapter 5, and the program specification is a subject of Chapter 6. That chapter also discusses the differences between our improved SLEX algorithm and the SLEX algorithm in (Abraham, Dolev, Herman, and Koll 2001). A formal correctness proof of our improved SLEX algorithm is presented in Chapter 7. That proof consists of a proof of the safety property, presented in Chapter 7.2, and a proof of the liveness property, presented in Chapter 7.3. Finally, Chapter 8 contains conclusion and discusses some future work.

CHAPTER 2

PROBLEM BACKGROUND AND RELATED WORK

In this chapter, we discuss the background of self-stabilizing ℓ-exclusion and some work related to mutual exclusion. In Section 2.1 we describe the types of interprocess communication employed in the shared memory model of process communication. In Section 2.2, we discuss the problem of mutual exclusion. The problem of ℓ-exclusion is then discussed in Section 2.3, and the concept of self-stabilization is introduced in Section 2.4. In Section 2.5, we focus on self-stabilizing mutual exclusion. Finally, Section 2.6 contains a discussion of self-stabilizing ℓ-exclusion.

2.1 Interprocess Communication

In 1986, Lamport introduced the notions of safe, regular, and atomic registers, see (Lamport 1986c; Lamport 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 the shared register. At higher levels of data sharing, particularly in distributed systems, the communicating processes may run at vastly different speeds. In that case, the wait and delay approach can lead to substantial degradation of concurrency and performance of the system. If 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 according to Lamport.

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

The next stronger possibility is the regular register. If a read operation is not concurrent with any write operation, the value most recently written to the register is returned as in the case of the safe register. If a read operation is concurrent with one or more write operations, then the read operation returns either the value that had been written to the register most recently before the read operation started, or one of the values written to the register concurrently with that read operation. As a consequence, if two consecutive read operations on a regular register are concurrent with two consecutive write operations on that register, it is possible that the earlier read operation returns the value written by the later write operation whereas the later read operation returns the value written by the earlier write operation.

The last possibility is the atomic register. It is a safe register in which all read and write operations behave as if no concurrent operations ever occur 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 total order. Consequently, once a read operation on an atomic register returns the value written by some write operation on that register, all subsequent read operations return either the value written by that write operation, or the value written by some later write operation.

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 access pattern, Lamport categorized registers as single/multiple-reader and single/multiple-writer. 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. The SLEX algorithm in this thesis uses single-writer multiple-reader regular registers.

2.2 Mutual Exclusion

The problem of mutual exclusion is one of the fundamental problems in computer science. Assume that there exist N sequential processes sharing a single resource. A process accesses the resource only when it executes part of its code known as the critical section. It is required that if a process executes its critical section, no other process is simultaneously in the critical section. In addition, every process that is attempting to enter its critical section must be able to do so within a finite amount of time. Nothing is assumed about the relative speeds of the processes.

In general, code of every program implementing mutual exclusion can be restructured to consist of the following four components: the trying section, the critical section, the exit section, and the remainder section. The trying section represents the code ensuring that no process enters the critical section if there is a process already executing the critical section. The exit section represents the code signaling that some process has just left the critical section. Finally, the remainder section represents the rest of the code that is unrelated to sharing the resource.

The first solution of the mutual exclusion problem for N processes was published by Dijkstra in (Dijkstra 1965). That solution is a generalization of Dekker's solution of mutual exclusion between two processes (Dijkstra 1968). Several other mutual exclusion algorithms were developed later, see, e.g., (Peterson 1981), (Lamport 1986b), and (Lycklama and Hadzilacos 1991).

2.3 ℓ-Exclusion

Fischer et al. 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, see (Fischer, Lynch, Burns, and Borodin 1979). If there are fewer than ℓ processes in their critical sections, it is possible for another process to enter its critical section. This generalization of the 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 the shared 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 a mutual exclusion algorithm.

Although the above solution correctly implements the ℓ-exclusion problem, it leads to a degradation of performance in those 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 cannot leave the queue, enter its critical section, and acquire another available instance of the resource. Thus the above solution lacks concurrency, a limitation particularly in distributed systems.

In (Fischer, Lynch, Burns, and Borodin 1979), the authors present several solutions of the ℓ-exclusion problem. Those solutions satisfy different concurrency and robustness properties, allowing to control the degradation of performance even when a limited number of processes fail.

2.4 Self-Stabilization

In 1973, Dijkstra introduced the concept of self-stabilization on the problem of mutual exclusion (Dijkstra 1974). Every mutual exclusion program satisfies the properties that at most one process is in the critical section at any moment of the program's computation, and that every process interested in entering the critical section eventually enters the critical section. If the shared memory is distributed, every process maintains a local copy of the shared data. If a participating process experiences some transient fault, data in the shared memory may become corrupt and the system may be put to an illegal (unstable) state. An illegal state is a state in which at least one of the properties of mutual exclusion does not hold. Once the program is in an illegal state, it usually remains in illegal states until the program is restarted from the initial state.

In (Dijkstra 1974), it was demonstrated that there exists a mutual exclusion program which does not need to be restarted after any occurrence of a transient fault. Instead, the program is always able to reach a legal state without outside intervention such that from that state onwards it remains in legal states and behaves as if no transient fault ever occurred. Thus the program is self-correcting in spite of the presence of transient faults. Later such self-correcting programs became known as self-stabilizing programs.

A self-stabilizing program can also be depicted as a program that may start its execution in an arbitrary initial state. (That initial state represents the state after the last transient fault occurred.) After the self-stabilizing program takes a finite number of steps, it reaches a legal state and thereafter it remains in legal states. The weak-fairness requirement is imposed on the program to ensure that the program eventually takes steps leading towards self-stabilization. Self-stabilizing programs are assumed to be non-terminating because otherwise it is not possible to ensure that any steps leading to self-stabilization are eventually taken. Recently, it has been shown that some message-passing programs can be both self-stabilizing and terminating (Arora and Nesterenko 2004).

In (Dijkstra 1974), Dijkstra presented three different self-stabilizing solutions of the mutual exclusion problem. Dijkstra's work did not gain any attention until 1984, when Lamport referred to self-stabilization in his invited address, see (Lamport 1984), and called it Dijkstra's probably most brilliant work.

2.5 Self-Stabilizing Mutual Exclusion

As mentioned in Section 2.4, the first self-stabilizing mutual exclusion algorithm was presented in (Dijkstra 1974). Dijkstra's solution used 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 was 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. In (Lamport 1986b), 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 requirements. For example, his self-stabilizing mutual exclusion algorithm that utilizes one shared bit of information per process satisfies the mutual exclusion property that at most one process is in the critical section at any moment of a computation, and the deadlock freedom property that if there exists a process trying to enter the critical section, then eventually some process enters the critical section. Lamport's three-bit solution satisfies both the properties of the one-bit solution, and in addition, it satisfies the strong-fairness property; that is, if a process is trying to enter its critical section, it eventually enters the critical section. Finally, if a program consists of N concurrent processes competing for a shared resource, the

N-bit solution given in (Lamport 1986b) satisfies all the requirements imposed on the three-bit solution and also the first-come, first-served property. That property states that processes enter the critical section in the order in which they became interested in entering the critical section.

2.6 Self-Stabilizing ℓ-Exclusion

Dijkstra's self-stabilizing solution of the mutual exclusion problem was generalized by Flatebo et al. in (Flatebo, Datta, and Schoone 1994). In that paper, the authors present a self-stabilizing ℓ-exclusion algorithm that uses 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 a token in the ring. Consequently, the system is not able to self-stabilize in case of a permanent failure (crash) of one of the processes, which is the main drawback of the solution. The prescribed ring communication topology is also a limitation.

Another algorithm implementing SLEX is presented in (Hadid 2002). That algorithm uses a tree-shaped network for the communication between processes. In addition, the algorithm cannot cope with crashed processes.

The first truly satisfactory self-stabilizing ℓ-exclusion algorithm was published by Abraham et al., see (Abraham, Dolev, Herman, and Koll 2001). (A preliminary version of the paper appears in (Abraham, Dolev, Herman, and Koll 1997)). Their ℓ-exclusion algorithm is self-stabilizing and tolerates crashes of (a limited number of) processes. It also does not prescribe any specific communication topology. In addition, the SLEX algorithm utilizes a set of single-writer multiple-reader regular registers as shared memory.

Recently, a solution to SLEX that utilizes self-stabilizing timestamps has been presented in (Abraham 2003). That solution was proposed in (Afek, Dolev, Gafni, Merritt, and Shavit 1994), but in 1994 no implementation of self-stabilizing timestamps was known. The timestamps-based solution satisfies the same safety and liveness properties as the improved SLEX algorithm presented in Chapter 4 of this thesis. On the other hand, the SLEX solution in (Abraham 2003) utilizes a set of single-writer multiple-reader atomic registers rather than the weaker single-writer multiple-reader regular registers, which are utilized in (Abraham, Dolev, Herman, and Koll 2001) and in the SLEX algorithm presented in this thesis. Also the self-stabilizing timestamps in (Abraham 2003) require, on average, almost five times more shared register memory than the algorithms in (Abraham, Dolev, Herman, and Koll 2001) and in this thesis.

CHAPTER 3

PRELIMINARIES

In this section we introduce some notions and notations used in the rest of this thesis. In Section 3.1 we describe the temporal logic used in our proof. The notion of a history (Herlihy and Wing 1990) is introduced in Section 3.2. Section 3.3 then describes the UNITY (Chandy and Misra 1988) representation of programs.

3.1 Linear–Time Temporal Logic

In our proof, properties of the SLEX program are formulated in Linear–Time Temporal Logic (LTL), see, e.g., (Manna and Pnueli 1983; Manna and Pnueli 1992; Manna and Pnueli 1995). The language of LTL consists of the following entities: variables, constants, propositions, functions, and predicates. The set of variables is partitioned into the set of program local variables, which may change during a program's execution, and a set of global variables, which are unchanged.

The following propositional logic operators are used in LTL: $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\equiv$, denoting respectively negation, conjunction, disjunction, implication, and equivalence. In addition, LTL uses the first-order predicate logic universal ($\forall$) and existential ($\exists$) quantifiers, which can be applied to global variables only. Formulas in LTL do not contain any occurrences of free variables. We adopt the abbreviation that every free variable in a formula is universally quantified.

Several modal operators are used in LTL. Those are $\bigcirc$, $\Box$, $\Diamond$, $\mathcal{U}$, and $\mathcal{W}$, called respectively *next*, *always*, *eventually*, *until*, and *weak-until* operators. The first three operators are unary; the until and weak-until operators are binary. Their semantics is explained below.

A program in LTL is modeled by a set of infinite computation sequences. A computation sequence σ is an infinite sequence of states $s_0, s_1, \ldots, s_i, \ldots$. Every state s_i assigns values to each program variable. Expression $\sigma^{(i)}$ denotes the suffix $s_i, s_{i+1}, \ldots$ of computation sequence σ.

We next describe the semantics of the modal operators. Let φ and ψ be temporal formulas and $\sigma = s_0, s_1, \ldots$ be a computation sequence. (1) Formula $\bigcirc\,\varphi$ is satisfied by computation σ iff φ is satisfied by $\sigma^{(1)}$. Intuitively, $\bigcirc\,\varphi$ holds in the first state of a computation iff φ holds in the next state of that computation. (2) Formula $\Box\varphi$ is satisfied by σ iff φ holds in every suffix $\sigma^{(i)}$, $i = 0, 1, \ldots$, of computation sequence σ. Thus $\Box\varphi$ intuitively means that formula φ always holds. (3) Formula $\Diamond\varphi$ is satisfied by σ iff there exists some suffix $\sigma^{(i)}$ such that φ is satisfied by $\sigma^{(i)}$. Intuitively, $\Diamond\varphi$ holds in σ iff φ eventually holds in σ. (4) Formula $\varphi\,\mathcal{U}\,\psi$ is satisfied by σ iff there exists some suffix $\sigma^{(k)}$ such that ψ is satisfied by $\sigma^{(k)}$ and formula φ is satisfied by every suffix $\sigma^{(i)}$, $i = 0, 1, \ldots, k-1$. Intuitively, $\varphi\mathcal{U}\psi$ holds iff eventually ψ holds and φ holds until ψ holds. (5) Formula $\varphi\,\mathcal{W}\,\psi$ is satisfied by σ iff $\Box\varphi$ is satisfied by σ or $\varphi\,\mathcal{U}\,\psi$ is satisfied by σ. Intuitively, $\varphi\,\mathcal{W}\,\psi$ holds iff φ holds unless ψ holds, but, in contrast to the $\mathcal{U}$ operator, ψ does not have to eventually hold. In the latter case, φ always holds.

For brevity, we use a version of the *freeze* quantifier ($\bullet$), introduced in the half-order modal logic, see (Henzinger 1990). Let y be a global variable, F be a total function on program states, and φ be a temporal formula. Formula $y = F \bullet \varphi$ is satisfied by computation sequence σ iff φ' is satisfied by σ, where φ' is the formula obtained from φ as follows: Every free occurrence of y in φ is replaced by the constant which is the value of function F evaluated in the first state of computation sequence

σ. Expression $y = F \bullet \varphi$ intuitively means that the value of variable y is permanently frozen (bound) to the value $F(s_0)$ and then φ' is evaluated. Note that $\Box y = F \bullet \varphi$ is not equivalent to $y = F \bullet \Box\varphi$, since in the first formula y is frozen to the current value of F in every suffix $\sigma^{(i)}$, $i = 0, 1, \ldots$, whereas in the latter formula y is frozen to the value of F in $\sigma^{(0)}$. Also note that from the semantics of the freeze quantification it follows that $y = F \bullet \neg\varphi$ is equivalent to $\neg(y = F \bullet \varphi)$. The freeze quantification $y = F \bullet \varphi$ is an abbreviation of $\forall y \cdot y = F \rightarrow \varphi$ and, equivalently, $\exists y \cdot y = F \wedge \varphi$ in the first-order predicate logic.

3.2 Histories

The SLEX algorithm employs the shared memory model of communication. The shared memory consists of a set of single-writer multiple-reader regular registers (Lamport 1986c; Lamport 1986d). A read operation on such a register X returns the value most recently written to X if that read operation is not concurrent with any write operation on X. If a read operation on register X is concurrent with some write operation on X, that read operation may return any value being concurrently written to X or the value most recently written to X before that read operation started.

In the semantics of the program, we utilize a *history* (Herlihy and Wing 1990) to model access to the regular registers. Intuitively, a history is a sequence of *events* recording the starts (invocations) and the conclusions (responses) of (read and write) operations performed by the program on its registers. Every event in a history refers to the register involved in the operation and to the process performing that operation. Some events also refer to the value that is read or written in that operation. For process P_i, register X, and value v, an event in a history is of the form

- $\langle i, \mathsf{inv}, \mathsf{rd}, X\rangle$ indicating an invocation of a read operation performed by process P_i on register X;

- $\langle i, \mathsf{res}, \mathsf{rd}, X, v\rangle$ indicating a response of a read operation performed by process P_i on register X, where v is the value of that response;

- $\langle i, \mathsf{inv}, \mathsf{wrt}, X, v\rangle$ indicating an invocation of a write operation performed by process P_i on register X, where v is the value being written; or

- $\langle i, \mathsf{res}, \mathsf{wrt}, X\rangle$ indicating a response of a write operation performed by process P_i on register X.

A history is a finite sequence of events. As usual, for history h, $|h|$ denotes the length of sequence h, i.e., the number of events in the history; and ε denotes the empty history, i.e., the sequence with zero elements. For natural number i, $1 \leq i \leq |h|$, $h[i]$ denotes the i^{th} event in history h; and if e is an event, $h \,^\wedge\, e$ denotes the history obtained by appending e to h.

An invocation $h[m]$ *matches* a response $h[n]$ if the invocation and the response denote the start and the conclusion of the same operation.

Definition 3.2.1. *Let h be a history and m and n be indices in h. Predicate $Match(h, m, n)$ holds iff $m < n$ and, for some process P_i, register X, value v, and for every value w, the following holds:*

$$\begin{aligned}
&(h[m] = \langle i, \mathsf{inv}, \mathsf{wrt}, X, v\rangle \wedge h[n] = \langle i, \mathsf{res}, \mathsf{wrt}, X\rangle \\
&\quad \wedge (m < k < n \rightarrow (h[k] \neq \langle i, \mathsf{inv}, \mathsf{wrt}, X, w\rangle \wedge h[k] \neq \langle i, \mathsf{res}, \mathsf{wrt}, X\rangle))) \\
\vee\ &(h[m] = \langle i, \mathsf{inv}, \mathsf{rd}, X\rangle \wedge h[n] = \langle i, \mathsf{res}, \mathsf{rd}, X, v\rangle \\
&\quad \wedge (m < k < n \rightarrow (h[k] \neq \langle i, \mathsf{inv}, \mathsf{rd}, X\rangle \wedge h[k] \neq \langle i, \mathsf{res}, \mathsf{rd}, X, w\rangle))) \ .
\end{aligned}$$

A history is *sequential* (Herlihy and Wing 1990) if it is either empty or if it is non-empty, starts with an invocation, every response in the history is immediately preceded by a matching invocation, and every invocation, except the first one, is immediately preceded by some response.

Every event in history h refers to some process. We define process P's sub-history of h as the projection of h onto events in h that refer to process P. We say that history h is *well-formed* if all processes' sub-histories of h are sequential, and if every read invocation on some register is preceded by some write response to that register in h. The latter requirement, not present in (Herlihy and Wing 1990), ensures that the value of every read operation is well-defined, see Chapter 5.

In a well-formed history, every response has its matching invocation, but some invocations may not have a matching response. A response with its matching invocation constitutes a complete operation; an invocation without a matching response constitutes an incomplete operation.

Definition 3.2.2. *Let h be a well-formed history, m, n be two indices in h, and $h[m]$ be some invocation.*

a) Pair $\langle m, n\rangle$ is a complete operation in h if $Match(h, m, n)$ holds.

b) Singleton $\langle m\rangle$ is an incomplete operation in h if $\neg Match(h, m, k)$ holds, for every index k in h.

By definition, every operation in a history is either complete or incomplete. Hereafter, predicate $Complete(h, op)$ denotes that operation op is a complete operation in h. If op is a complete operation in h, $Start(h, op)$ denotes the index of op's invocation and $Concl(h, op)$ denotes the index of op's response in h. Thus, if $\langle m, n\rangle$ is a complete operation in h, $Start(h, op) = m$ and $Concl(h, op) = n$.

Sometimes we also refer to the type of an operation. Intuitively, the type of an operation characterizes the process which performs that operation, the kind of that operation, and the register involved in that operation. More precisely, let $op = \langle m, n\rangle$ or $op = \langle m\rangle$ be an operation in history h, P_i be a process, and X be a register. We say op is a read operation on register X by process P_i, denoted by $OpType(h, op) = \langle i, \mathsf{rd}, X\rangle$, if $h[m] = \langle i, \mathsf{inv}, \mathsf{rd}, X\rangle$. If $h[m] = \langle i, \mathsf{inv}, \mathsf{wrt}, X, v\rangle$, for some value v, we say

op is a write operation on register X and we denote this by $OpType(h, op) = \langle \mathsf{wrt}, X\rangle$. The latter type does not explicitly specify the process which performs that operation. That process is determined by register X because X is a single-writer register.

Every write operation involves some value written to the register, and every complete read operation involves some value read from the register. The value involved in a write operation is recorded in the invocation of that operation whereas the value involved in a read operation is recorded in the response of that read operation. Thus the value involved in a read operation can be determined only when that read operation is complete.

Definition 3.2.3. *Let h be a well-formed history, and let $op = \langle m, n\rangle$ or $op = \langle m\rangle$ be an operation in h. If $h[m] = \langle i, \mathsf{inv}, \mathsf{wrt}, X, v\rangle$, or if op is a complete read operation and $h[n] = \langle i, \mathsf{res}, \mathsf{rd}, X, v\rangle$, then the value of operation op, denoted by $Val(h, op)$, is v. Otherwise, i.e., if op is an incomplete read operation, $Val(h, op)$ is undefined.*

In our correctness proof we use the notions of one operation preceding another operation, one operation being concurrent with another operation, one operation immediately preceding another operation, the last write operation to a register, and the last read operation on a register performed by some process.

Definition 3.2.4. *Let h be a well-formed history. Let $op = \langle m, n\rangle$ or $op = \langle m\rangle$ be some operation in h, and let $op' = \langle m', n'\rangle$ or $op' = \langle m'\rangle$ be an operation in h different from op.*

a) Operation op precedes op' in h, denoted by $op \prec op'$, if $op = \langle m, n\rangle$ and $n < m'$ are satisfied.

b) Operation op is concurrent with op' in h, denoted by $op \parallel op'$, if $\neg(op \prec op' \vee op' \prec op)$ is satisfied.

c) Operation op immediately precedes op' in h, denoted by $op \sqsubset op'$, if $op \prec op'$

holds and if, for every operation op''*,* $op \prec op'' \prec op'$*,* $\mathit{OpType}(h, op'') \neq \mathit{OpType}(h, op)$ *holds.*

d) *Operation* op *is the last (possibly incomplete) write operation in* h *on register* X*, denoted by* $op = \mathit{LastWrt}(h, X)$*, if* $\mathit{OpType}(h, op) = \langle \mathsf{wrt}, X \rangle$*, and every operation* op'*,* $op' \neq op$*, of type* $\langle \mathsf{wrt}, X \rangle$ *precedes* op*.*

e) *Operation* op *is the last (complete) read operation in* h *on register* X *by process* P_i*, denoted by* $\mathit{LastRd}(h, op, i, X)$*, if* op *is complete,* $\mathit{OpType}(h, op) = \langle i, \mathsf{rd}, X \rangle$*, and every operation* op'*,* $op' \neq op$*, of type* $\langle i, \mathsf{rd}, X \rangle$ *precedes* op*.*

Intuitively, operation op precedes op' in history h if op had completed before op' started. (In this case, operation op' may be either a complete or an incomplete operation.) Two operations are concurrent in h if neither of the operations precedes the other operation in h. Operation op immediately precedes op' in h if op is the most recent operation of type $\mathit{OpType}(h, op)$ that precedes op' in the history. (Note that in a well-formed history every process's sub-history is sequential. Therefore, in a well-formed history, for two consecutive operations of the same type, one of the operations precedes the other operation.) The last (possibly incomplete) write operation is unique and always defined in a well-formed history. Thus $\mathit{LastWrt}(h, X)$ is a total function. The last (complete) read operation on a register may not exist. If in a well-formed history there exists a complete read operation on some register by a process, then the last (complete) read operation on that register by that process is defined and unique.

In our correctness proof, we also identify those write operations that may influence the outcome of a read operation.

Definition 3.2.5. *Let* h *be a well-formed history,* r *be a complete read operation in* h*, and* w *be a write operation in* h*. We define*

$$\mathit{ValSrc}(h, r, w) \equiv \exists i \exists X \cdot \mathit{OpType}(h, r) = \langle i, \mathsf{rd}, X \rangle \wedge \mathit{OpType}(h, w) = \langle \mathsf{wrt}, X \rangle$$

$$\wedge\, \mathit{Val}(h,r) = \mathit{Val}(h,w) \wedge (w \sqsubset r \vee w \parallel r) \ \ .$$

If predicate $\mathit{ValSrc}(h,r,w)$ *holds, we say that write operation* w *is a value source of read operation* r *in history* h.

Intuitively, write operation w is a value source of read operation r in history h if operations r and w access the same register, have the same value, and w either immediately precedes r or w is concurrent with r in h.

For readability, we adopt the following abbreviations: We do not mention history parameter h in a predicate if h is the history component of the state in which that predicate is being evaluated. Whenever we refer to the most recent value of a register, we denote that value by the name of that register typeset using a **bold face** font. Thus, for the most recently written value of register X, we write $\mathbf{X}$ instead of $\mathit{Val}(\mathit{LastWrt}(\mathsf{X}))$.

3.3 UNITY Programs

UNITY (Chandy and Misra 1988) is a guarded command language particularly suitable for a uniform representation of sequential and concurrent programs. In UNITY, every command (called the action) of a program is guarded by a quantifier-free predicate, called the guard. An action is an atomically executed statement represented by a state transformation function.

Let G be a guard and τ be an action. In UNITY, transition $G \longrightarrow \tau$ expresses that action τ is guarded by guard G. Every UNITY program is then represented by some finite set of transitions

$$\begin{array}{lcl} G_1 & \longrightarrow & \tau_1 \ \ , \\ G_2 & \longrightarrow & \tau_2 \ \ , \\ \vdots & \vdots & \vdots \\ G_n & \longrightarrow & \tau_n \ \ , \end{array}$$

where G_i are guards and τ_i are actions, for $i = 1, 2, \ldots, n$.

In any state of a UNITY program, only transitions that are enabled in that state may be taken. A transition is enabled in a state if the guard of that transition is satisfied in that state; otherwise, the transition is disabled in that state. If several transitions are enabled in a state, the choice of the taken transition is non-deterministic.

CHAPTER 4

PROGRAM DESCRIPTION

The self-stabilizing ℓ-exclusion program $\mathcal{S}$ is a system of $N > 1$ concurrent processes $P_1, \ldots, P_N$. Every process executes a copy of the program in Figures 4.1 and 4.2, and utilizes a set of (local) variables and a set of (shared) single-writer multiple-reader regular registers. For convenience, register names are typeset using a Sans Serif font. In addition, names of registers that may be modified (written) by process P_i are subscripted by index i. In the program, (local) variables are typeset using *italics* and have not been subscripted. When needed for clarity, names of process P_i's variables will be subscripted by index i.

To ensure the safety property that at most $\ell \leq N$ processes can be in their critical sections simultaneously, the program employs a set of boolean registers $\mathsf{TRY_i}$. Register $\mathsf{TRY_i}$ is written value *true* when process P_i attempts to enter its critical section, and *false* when P_i leaves its critical section. Every process P_i keeps track (in variable B_i) of all processes about which P_i assumes to be in the critical section. Process P_i enters its critical section only if $|B_i| \leq \ell$, where $|B_i|$ denotes the cardinality of set B_i.

It is possible that process P_i reads register $\mathsf{TRY_j}$, for some process P_j different from P_i, always at the moments when $\mathsf{TRY_j}$ is *true*, even if the value recorded in $\mathsf{TRY_j}$ was set to *false* in the meantime. Consequently, from P_i's point of view, process P_j is continuously in its critical section. This may lead to starvation of process P_i.

```
        function $getTry()$
$\ell_{8.1}$:   for $j := 1$ to $N$ do begin
$\ell_{8.2}$:       read$(t[j] := \mathsf{TRY_j})$;
$\ell_{8.3}$:       read$(x[j] := \mathsf{X_j})$;
$\ell_{8.4}$:       $a[j] := x[j] \vee t[j]$;
$\ell_{8.5}$:       if $a[j]$ then
$\ell_{8.6}$:           read$(ord[j] := \mathsf{ORD_j})$
$\ell_{8.7}$:   end;
$\ell_{8.8}$:   $p := \pi(choice(ord, \gamma(a)), i)$;
$\ell_{8.9}$:   $A := \{j \mid j \neq i \wedge t[j] \wedge (j \in p \vee \exists i' \cdot (i' \in p \wedge \neg dominates(v, j, i')))\}$;
$\ell_{8.10}$:  return $|A| < \ell$
        end
```

Figure 4.1: Code of function *getTry*().

To ensure liveness, some processes will be excluded from set B_i. Specifically, every process will be excluded from B_i which is 'much faster' than process P_i. (The notion of 'much faster' is made precise below.) Of course, if the above mentioned processes are excluded from B_i, the safety property remains preserved.

In order to determine whether one process is 'much faster' than another one, program $\mathcal{S}$ employs an array of *vectors*. Vector $\mathsf{VEC_i}$ is a register consisting of N elements. Element $\mathsf{VEC_i}[i]$, called process *P_i's color*, is some natural number in $\{1, \ldots, 2N\}$. Element $\mathsf{VEC_i}[j]$, for every process P_j, $j \neq i$, is a pair of colors referred to as *first* and *second*. Every process P_i maintains (local) copies of all vectors $\mathsf{VEC_j}$ as rows $v_i[j]$ in matrix v_i.

The new value of register $\mathsf{VEC_i}$ is computed by function *report*(v, i), which returns some vector r satisfying the following: Process P_i's color in r is some color different from every color in column i of matrix v. (Such a color exists since $2N$ colors are

```
        repeat forever
$\ell_1$:      repeat
$\ell_2$:           write($X_i := true$);
$\ell_3$:           for $j := 1$ to $N$ do begin
$\ell_4$:                read($v[j] := VEC_j$)
$\ell_5$:           end;
$\ell_6$:           $vec := report(v, i)$;
$\ell_7$:           read($old\_try := TRY_i$);
$\ell_8$:           $try := getTry()$;
$\ell_9$:           write($TRY_i := try$);
$\ell_{10}$:          if $try \wedge \neg old\_try$ then
$\ell_{11}$:               write($VEC_i := vec$)
$\ell_{12}$:      until $try$;
$\ell_{13}$:      for $j := 1$ to $N$ do begin
$\ell_{14}$:           read($v[j] := VEC_j$);
$\ell_{15}$:           read($t[j] := TRY_j$)
$\ell_{16}$:      end;
$\ell_{17}$:      $B := \{j \mid t[j] \wedge \neg dominates(v, j, i)\}$;
$\ell_{18}$:      if $|B| > \ell$ then goto $\ell_1$;
$\ell_{19}$:      Critical Section;
$\ell_{20}$:      write($TRY_i := false$);
$\ell_{21}$:      write($X_i := false$);
$\ell_{22}$:      $row := change(ord, \gamma(a), i)$;
$\ell_{23}$:      write($ORD_i := row$);
$\ell_{24}$:      Remainder Section
        end
```

Figure 4.2: Program executed by process P_i, for $i = 1, \ldots, N$.

available and at most $2N-1$ colors are used in column i.) For every process P_j, $j \neq i$, process P_j's color (recorded in $v[j][j]$) is shifted to the *first* color of the pair $r[j]$. And the original *first* color (recorded in $v[i][j].\mathit{first}$) is shifted to the *second* color of the pair $r[j]$. Thus, the following holds for vector $r = \mathit{report}(v,i)$ and for every process P_j, $j \neq i$: $r[i] \neq v[i][i] \wedge r[i] \neq v[j][i].\mathit{first} \wedge r[i] \neq v[j][i].\mathit{second} \wedge r[j].\mathit{first} = v[j][j] \wedge r[j].\mathit{second} = v[i][j].\mathit{first}$.

Every time process P_i attempts to enter its critical section, an updated vector is written to register $\mathsf{VEC_i}$. Therefore, for every process P_j, P_j's color is shifted to the *first* and *second* colors of $\mathsf{VEC_i}[j]$. Consequently, if process P_i updates its register $\mathsf{VEC_i}$ 'much faster' than process P_j, the colors $\mathsf{VEC_i}[j].\mathit{first}$, $\mathsf{VEC_i}[j].\mathit{second}$, and $\mathsf{VEC_j}[j]$ will be the same. Whether P_i is 'much faster' than P_j is determined by means of function $\mathit{dominates}(v,i,j)$, which returns *true* iff $i \neq j \wedge v[j][j] = v[i][j].\mathit{first} = v[i][j].\mathit{second}$ holds. We then say that process P_i dominates process P_j.

It could be the case that more than ℓ processes P_i simultaneously update their colors in $\mathsf{VEC_i}[i]$, hence making every P_i's color different from every other color in column i. Therefore, no process dominates any of the above mentioned processes P_i. Consequently, set B_i, for every process P_i, will contain at least ℓ processes, which may lead to starvation. To ensure liveness, at most ℓ processes are selected that may simultaneously attempt to enter their critical sections and write *true* to their register $\mathsf{TRY_i}$. This selection is based on process priorities, as described next.

Program $\mathcal{S}$ employs a set of registers $\mathsf{ORD_i}$ to determine a process's priority. Every register $\mathsf{ORD_i}$ is an array of N bits and forms the i^{th} row of a (square) bit-matrix. A (local) copy of the bit-matrix is maintained in variable ord_i, for every process P_i. A process has the k^{th} highest priority if it is selected in the k^{th} column of the bit-matrix. The selection process is described next.

In the bit-matrix, each column k is viewed as a carousel. Every process P_i's seat in that carousel carries either 0 or 1. Some of those seats are marked inactive. Seats marked inactive belong to the processes currently not wishing to enter nor being in their critical sections. Number 0 in P_i's seat indicates that process P_i has visited its critical section an even number of times, whereas number 1 indicates an odd number of visits. If all active seats carry the same value, the first active process in column k is selected as the k^{th} highest priority process. Otherwise, the minimal process among all active processes carrying a value different from the value of the first active process is selected. After process P_i leaves its critical section, the parity recorded in P_i's seat is updated. That updated parity equals the parity recorded in the previous active process's seat unless all seats belonging to active processes carry the same parity. In that case, the first active process inverts its parity. Thus the number of consecutive seats with the same parity increases as processes visit their critical sections until all seats carry the same value.

If more than ℓ processes are attempting to enter the critical section, some processes will set their register $\mathsf{TRY_i}$ to *false* to avoid deadlock. Therefore, register $\mathsf{TRY_i}$ no longer suffices as an indicator that process P_i is wishing to enter its critical section. For that purpose the program employs a set of boolean registers $\mathsf{X_i}$.

Process P_i is active if P_i indicates its wish (by writing *true* to $\mathsf{X_i}$) or attempt (by writing *true* to $\mathsf{TRY_i}$) to enter its critical section. Process P_i is active in column k (of bit-matrix ord_i) if P_i is active and it has not been selected in any column preceding column k.

We now formalize the selection process described above. Let a be a non-empty set of active process indices. Function $\mathit{First}(a) = \min(a)$ denotes the first process in a, and $\mathit{Last}(a) = \max(a)$ denotes the last process in a. Let ord be the local copy of the bit-matrix of registers $\mathsf{ORD_i}$. Process P_i is selected in column k of bit-matrix ord if P_i is the minimal active process in column k that has the same value as the last active process in that column. Formally, $\mathit{Select}(ord, a, k) = \min\{i \mid i \in a \wedge ord[i][k] = ord[\mathit{Last}(a)][k]\}$.

Function *Select* is utilized in function *choice*, which determines the process priorities. Intuitively, the process selected in the first column of a bit-matrix has the highest priority; the process selected in the second column has the second highest priority; and so forth. For set a of active processes and a local copy ord of the bit-matrix ORD, function $choice(ord, a)$ returns the sequence of active processes sorted in decreasing order with respect to their priorities. That sequence s has length $|a|$, and for every k, $1 \leq k \leq |a|$, element $s[k]$ equals $Select(ord, a \setminus \rho(s, k), k)$. Here, function $\rho(s, k) = \{s[i] \mid i < k\}$ determines the set of processes that were selected earlier than in column k. Processes in $\rho(s, k)$ are then removed from the set of active processes in column k and all subsequent columns.

One of the arguments of function *choice* is a set of active processes. That set is computed by function γ. Function γ takes as its argument an array a of boolean values. Only if element $a[i]$ equals $true$, process P_i is included in the set of active processes. Formally, $\gamma(a) = \{i \mid a[i] = true\}$.

After process P_i leaves its critical section, register $\mathsf{ORD_i}$ is updated with the new parity of P_i's visits in the critical section. The new value of $\mathsf{ORD_i}[k]$ depends on the parity of visits of the previous active process in column k. Recall that every column is viewed as a carousel. Therefore, the first active process in column k is preceded by the last active process in column k. Every process should visit the critical section the same parity of times. After a process leaves its critical section, the sequence of consecutive processes with the same parity is extended. Hence, after process P_i leaves its critical section, the new value recorded in $\mathsf{ORD_i}[k]$ equals the value recorded in $\mathsf{ORD_j}[k]$, for previous active process P_j, unless P_j is the first active process in column k. In that case, a new round of visits starts and the new value recorded in $\mathsf{ORD_i}[k]$ is set to differ from the last active process's value in column k. If there is no active process in column k, the new value of $\mathsf{ORD_i}[k]$ is unimportant and reset to 0. Due to the parity copying, if some process other than P_i is selected in column k, that process remains selected in that column even after $\mathsf{ORD_i}$ is updated. Consequently, priorities of processes selected before P_i will not be lowered by P_i's activity.

We now precisely describe how the new value of register $\mathsf{ORD_i}$ is determined. Let a be a set of active processes. If a is non-empty, then function $\mathit{Prev}(a, i) = \max\{j \mid j \in a \wedge (j < i \vee i \leq \mathit{First}(a))\}$ determines the process that immediately precedes process P_i in a carousel. Let ord be a local copy of the bit-matrix ORD and k be a column index in ord. Function $\mathit{Prior}(\mathit{ord}, a, i, k)$ determines the new value of register $\mathsf{ORD_i}$ in column k. It is defined as follows: If a is empty, $\mathit{Prior}(\mathit{ord}, a, i, k)$ equals 0; if $\mathit{Prev}(a, i) < i$ holds, $\mathit{Prior}(\mathit{ord}, a, i, k)$ equals $\mathit{ord}[\mathit{Prev}(a, i)][k]$; otherwise, i.e., if $\mathit{Prev}(a, i) \geq i$, $\mathit{Prior}(\mathit{ord}, a, i, k)$ equals $1 - \mathit{ord}[\mathit{Prev}(a, i)][k]$. Finally, function $\mathit{change}(\mathit{ord}, a, i)$ determines the new value of register $\mathsf{ORD_i}$. Intuitively, the set of processes active in every column k is first determined and then function Prior is used to determine the new value of $\mathsf{ORD_i}[k]$. Thus, $\mathit{change}(\mathit{ord}, a, i) = o$ iff $o[k] = \mathit{Prior}(\mathit{ord}, a \setminus \rho(\mathit{choice}(\mathit{ord}, a), k), i, k)$, for every column k.

Once process priorities are computed, the set of active processes that have a higher priority than process P_i is computed. Assume that s is a sequence of processes ordered in decreasing order with respect to their priorities. Then function $\pi(s, i) = \{s[j] \mid 1 \leq j \leq |s| \wedge (1 \leq k \leq j \rightarrow s[k] \neq i)\}$ determines the set of processes that have a higher priority than process P_i. Those processes are then recorded in variable A_i. Only if $|A_i| < \ell$ holds, register $\mathsf{TRY_i}$ is set to true and P_i attempts to enter its critical section. Otherwise, P_i remains actively waiting until its priority increases.

In the states following a transient fault, it is possible that registers $\mathsf{ORD_i}$ indicate that ℓ non-crashed processes have a higher priority than some process crashed in the critical section. (If no failure occurs, all active crashed processes have eventually higher priority than every non-crashed process.) Thus those ℓ non-crashed processes determine that $|A_i| < \ell$ holds and they set their registers $\mathsf{TRY_i}$ to true. This again may lead to starvation. To ensure liveness, additional processes are included in set

```
    function $getTry()$
$\ell_{8.1}$:    for $j := 1$ to $N$ do begin
$\ell_{8.2}$:        read($t[j] := \mathsf{TRY_j}$);
$\ell_{8.3}$:        read($x[j] := \mathsf{X_j}$);
$\ell_{8.4}$:        if $x[j]$ then
$\ell_{8.5}$:            read($ord[j] := \mathsf{ORD_j}$)
$\ell_{8.6}$:    end;
$\ell_{8.7}$:    $p := \pi(choice(ord, \gamma(x)), i)$;
$\ell_{8.8}$:    $A := \{j \mid j \neq i \wedge (j \in p \vee (t[j] \wedge \exists i' \cdot (i' \in p \wedge \neg dominates(v, j, i'))))\}$;
$\ell_{8.9}$:    return $|A| < \ell$
    end
```

Figure 4.3: Code of the original function *getTry*().

A_i. Intuitively, the processes which are possibly crashed in the critical section and have a lower priority than process P_i are included in set A_i. More precisely, every process identified by P_i as in the critical section and not dominating to any process with a higher priority than P_i is included in A_i.

The SLEX program in the current paper is a slightly improved version of the SLEX program in (Abraham, Dolev, Herman, and Koll 2001). Every register $\mathsf{ORD_j}$ consists of N bits, and not only of ℓ bits as in (Abraham, Dolev, Herman, and Koll 2001). In addition, function *getTry* in Figure 4.1 differs from function *getTry* in (Abraham, Dolev, Herman, and Koll 2001), see Figure 4.3. In (Abraham, Dolev, Herman, and Koll 2001), process P_j is active if the value recorded in register $\mathsf{X_j}$ is *true*. We identify process P_j as active if $\mathsf{X_j}$ = *true* or if the value recorded in register $\mathsf{TRY_j}$ is *true*. Because of this modification (together with the increased size of registers $\mathsf{ORD_j}$), our improved SLEX program is able to assign a priority to every process crashed in the critical section or crashed trying to enter the critical section, i.e., with $\mathsf{TRY_j}$ = *true* or $\mathsf{X_j} = \textit{true}$.

To ensure liveness, the counting of higher priority processes (in set A_i) has also been modified. Our modification of function *getTry* preserves the property of the original function *getTry* in (Abraham, Dolev, Herman, and Koll 2001) that eventually at most ℓ processes have their registers $\mathsf{TRY_i}$ equal *true*, provided that less that ℓ processes are crashed in the critical section. To ensure that, we do not include all processes with a higher priority than process P_i in set A_i. Instead, a higher priority process is included in A_i only if it is identified as in the critical section. That is, only the higher priority processes with $\mathsf{TRY_j} = \textit{true}$ are included in A_i. In other words, processes crashed with $\mathsf{X_j} = \textit{true}$ and $\mathsf{TRY_j} = \textit{false}$ are excluded from A_i.

CHAPTER 5

PROGRAM MODEL AND SEMANTICS

In our proof, the program in Figures 4.1 and 4.2 is represented by a UNITY-like program, see Section 3.3. Obtaining the UNITY representation of program $\mathcal{S}$ from its description in Figures 4.1 and 4.2 is straightforward. That UNITY representation can be found in Appendix A.

A *state* of program $\mathcal{S}$ is represented by a pair $\langle \sigma, h \rangle$, where σ is a function mapping all variables into their domain of interpretation, and h is a well-formed history. In function σ, in addition to the program's variables, we introduce variable loc_i and auxiliary variable $crashed_i$, for every process P_i. Variable loc_i models P_i's program counter; variable $crashed_i$ takes boolean values and holds iff P_i is crashed. If process P_i is crashed, the guards of all P_i's transitions are permanently disabled.

The semantics of the program (Stoy 1977; Schmidt 1986) are defined as usual with the exception of the critical and remainder sections, and the read and write operations on regular registers. Although executing each of the critical and remainder sections may consist of several steps, this is irrelevant for the correctness proof and therefore not modeled. Therefore, for every process, each of the above sections is modeled by one atomic transition. We assume that in those sections there are no references to any registers. Consequently, a process that remains forever in its critical or its remainder section is crashed.

We next define the semantics of commands $\mathsf{read}(y{:=}X)$ and $\mathsf{write}(X{:=}e)$, where X

is some register, y is some variable, and e is some expression. Each of these commands is modeled by two atomic transitions: $\mathsf{invoke\_read}(X)$ and $\mathsf{respond\_read}(y, X)$, resp. $\mathsf{invoke\_write}(X, e)$ and $\mathsf{respond\_write}(X)$. Here, $\mathsf{invoke\_read}$ (or $\mathsf{invoke\_write}$) denotes the start of a read (or write) command, and $\mathsf{respond\_read}$ (or $\mathsf{respond\_write}$) denotes the conclusion of the corresponding command. Although execution of the actual command may consist of several steps, again, this is irrelevant for the correctness proof and therefore these steps are not modeled.

Execution of command $\mathsf{invoke\_write}(X, e)$ by process P_i in state $\langle \sigma, h \rangle$ appends event $\langle i, \mathsf{inv}, \mathsf{wrt}, X, [\![e]\!](\sigma) \rangle$, where $[\![e]\!](\sigma)$ denotes the value of expression e in σ, to history h and updates the value of program counter loc_i. Formally,

$$[\![\mathsf{invoke\_write}(X, e)]\!](\langle \sigma, h \rangle) = \langle \sigma\{{}^{next(loc_i)}/_{loc_i}\}, h \wedge \langle i, \mathsf{inv}, \mathsf{wrt}, X, [\![e]\!](\sigma) \rangle \rangle \ .$$

In the above, expression $next(loc_i)$ denotes the location after the action with entry point loc_i has been taken. (Because processes are sequential and deterministic, see Figures 4.1 and 4.2, expression $next(loc_i)$ is well-defined.) Expression $\sigma\{{}^{next(loc_i)}/_{loc_i}\}$ denotes the usual state variant: For variables y, z and expression e, we define

$$\sigma\{{}^{e}/_{y}\}(z) = \begin{cases} [\![e]\!](\sigma) & \text{if } y \equiv z; \\ \sigma(z) & \text{otherwise.} \end{cases}$$

Execution of $\mathsf{respond\_write}(X)$ by P_i in state $\langle \sigma, h \rangle$ appends event $\langle i, \mathsf{res}, \mathsf{wrt}, X \rangle$ to history h and updates the value of program counter loc_i. Formally,

$$[\![\mathsf{respond\_write}(X)]\!](\langle \sigma, h \rangle) = \langle \sigma\{{}^{next(loc_i)}/_{loc_i}\}, h \wedge \langle i, \mathsf{res}, \mathsf{wrt}, X \rangle \rangle \ .$$

Execution of $\mathsf{invoke\_read}(X)$ by P_i in state $\langle \sigma, h \rangle$ appends event $\langle i, \mathsf{inv}, \mathsf{rd}, X \rangle$ to history h and updates the value of program counter loc_i. Formally,

$$[\![\mathsf{invoke\_read}(X)]\!](\langle \sigma, h \rangle) = \langle \sigma\{{}^{next(loc_i)}/_{loc_i}\}, h \wedge \langle i, \mathsf{inv}, \mathsf{rd}, X \rangle \rangle \ .$$

Finally, we define the semantics of $\mathsf{respond\_read}$. If the read operation on register X by process P_i is not concurrent with any write operation on X, then the value of that read operation equals the value of the last write operation to X. (In a well-formed history such a write operation always exists.) Otherwise, the value of that read operation equals the value of any of the concurrent write operations on X or the value of the write operation to register X which immediately precedes that read operation. Thus, if y is a local variable, the semantics of $\mathsf{respond\_read}(y, X)$ executed by process P_i in state $\langle \sigma, h \rangle$ is defined as

$$[\![\mathsf{respond\_read}(y, X)]\!](\langle \sigma, h \rangle) = \langle \sigma\{{}^{next(loc_i)}/{}_{loc_i}, {}^{v}/{}_{y}\}, h \mathbin{\hat{}} \langle i, \mathsf{res}, \mathsf{rd}, X, v \rangle \rangle \ ,$$

where v is the value of some write operation to register X which immediately precedes or is concurrent with the (incomplete) read operation on register X by process P_i. Execution of $\mathsf{respond\_read}$ also updates the value of program counter loc_i and records the value of that read operation in variable y.

Value v can be precisely characterized as follows: Let $k = \max\{r \mid 1 \le r \le |h| \wedge h[r] = \langle i, \mathsf{inv}, \mathsf{rd}, X \rangle\}$ be an index of the event in history h. Event $h[k]$ then corresponds to the invocation matching the response of the current read operation on register X by process P_i. Let $m = \max\{w \mid \exists j \cdot 1 \le w < k \wedge h[w] = \langle j, \mathsf{res}, \mathsf{wrt}, X \rangle\}$ be an index in h. Then $h[m]$ is the event recording the conclusion of the write operation on X immediately preceding the current read operation on X by P_i; and $n = \max\{w \mid \exists j \exists y \cdot 1 \le w < m \wedge h[w] = \langle j, \mathsf{inv}, \mathsf{wrt}, X, y \rangle\}$ is the index in h such that event $h[n]$ is the invocation matching $h[m]$. Value v is then any value written by that write operation $\langle n, m \rangle$ or by any subsequent (and therefore concurrent with the current read operation) write operation on X. That is, $\exists w \cdot n \le w \le |h| \wedge \exists j \cdot h[w] = \langle j, \mathsf{inv}, \mathsf{wrt}, X, v \rangle)$ holds.

The meaning of program $\mathcal{S}$ is defined in terms of *computation sequences.* A computation sequence is an infinite sequence $\langle \sigma_0, h_0 \rangle \xrightarrow{\tau_0} \langle \sigma_1, h_1 \rangle \xrightarrow{\tau_1} \langle \sigma_2, h_2 \rangle \xrightarrow{\tau_2} \langle \sigma_3, h_3 \rangle \xrightarrow{\tau_3} \ldots$ such that, for all $i \ge 0$, the following hold:

- $\langle \sigma_i, h_i \rangle$ are program states,
- τ_i are atomically executed transitions (or actions),
- in $\langle \sigma_i, h_i \rangle$, transition τ_i is enabled, and
- state $\langle \sigma_{i+1}, h_{i+1} \rangle$ is the result of executing transition τ_i in state $\langle \sigma_i, h_i \rangle$.

We assume that every computation sequence of program $\mathcal{S}$ is weakly-fair (Francez 1986). That is, if some transition is enabled from some point onwards in a computation sequence, then that transition is taken infinitely often in that computation sequence. Furthermore, we assume that in state $\langle \sigma_0, h_0 \rangle$ the initial condition described in Section 6.1 is satisfied.

CHAPTER 6

PROGRAM SPECIFICATION

6.1 Initial Condition

The *initial state* $\langle \sigma_0, h_0 \rangle$ of program $\mathcal{S}$ models the situation after the last crash or transient fault has occurred. Such a fault may lead to the assignment of an arbitrary value to any program variable or register. To model such assignments, the initial condition does not restrict the values of program variables nor the values recorded in registers. The initial condition restricts the format and ordering of the initial events in the history to ensure that histories are well-formed.

In $\langle \sigma_0, h_0 \rangle$, σ_0 maps every variable to some admissible value in the variable's domain of interpretation, except for $\mathit{crashed}_i$, which is *true* if P_i is crashed; *false* otherwise. History h_0 is obtained from the empty history as follows: For every register X written by process P_i and some arbitrary admissible value v of X, consecutive events $\langle i, \mathsf{inv}, \mathsf{wrt}, X, v \rangle$ and $\langle i, \mathsf{res}, \mathsf{wrt}, X \rangle$ are appended to h_0. In addition, if in σ_0 process P_i is not crashed and program counter loc_i indicates that P_i is at a location where a read or a write operation on register X is in progress, the following event is appended to h_0: If P_i is at a location where a read operation is in progress, event $\langle i, \mathsf{inv}, \mathsf{rd}, X \rangle$ is appended to h_0; and if P_i is at a location where a write operation is in progress, event $\langle i, \mathsf{inv}, \mathsf{wrt}, X, v \rangle$ is appended to h_0, where v is the value of the preceding write operation to register X in h_0.

6.2 SLEX Specification

We next formalize the notions of a process being in the critical section, a process being crashed in the critical section, and a process being crashed while trying to enter the critical section. Hereafter, expression $[\ell_a, \ell_b]$ denotes the set of locations ℓ_i, $a \leq i \leq b$, in program $\mathcal{S}$.

Definition 6.2.1. *Let P_i be a process. In any state of program $\mathcal{S}$, we say that*

a) P_i is in the critical section, denoted by InCS_i, if the most recently written value to register $\mathsf{TRY_i}$ *is true, and either program counter loc_i is at a location in $[\ell_{19}, \ell_{20}]$ or P_i is crashed;*

b) P_i is crashed in the critical section, denoted by $\mathit{CrashedInCS}_i$, if it is in the critical section and it is crashed; and

c) P_i is crashed trying to enter the critical section, denoted by $\mathit{CrashedTrying}_i$ if it is crashed, the most recently written value to register $\mathsf{X_i}$ *is true, and the most recently written value to register* $\mathsf{TRY_i}$ *is false.*

Program $\mathcal{S}$ is to satisfy the following safety property: At most ℓ processes are simultaneously in their critical sections from some moment onwards, provided that at most ℓ processes are crashed in the critical section. The program is to satisfy the following liveness property: Every non-crashed process P_i is infinitely often in its critical section, provided that less than ℓ processes are crashed in the critical section.

Definition 6.2.2 (Formal Specification)**.** *Program $\mathcal{S}$ is to satisfy*

a) **Safety Property:**

$$|\{j \mid \mathit{CrashedInCS}_j\}| \leq \ell \rightarrow \Diamond\Box\left(|\{i \mid \mathit{InCS}_i\}| \leq \ell\right) \quad .$$

b) **Liveness Property:**

$$(|\{j \mid \mathit{CrashedInCS}_j\}| < \ell \wedge \neg \mathit{crashed}_i) \rightarrow \Box\Diamond \mathit{InCS}_i \quad .$$

The SLEX algorithm in (Abraham, Dolev, Herman, and Koll 2001), see Figure 4.3, satisfies the above safety property; however, it does not satisfy the above liveness property. Consider, for example, a computation sequence in which ℓ processes are crashed trying to enter the critical section and no other process is crashed. That means that ℓ processes are crashed with $\mathsf{X_j} = \mathit{true}$ and $\mathsf{TRY_j} = \mathit{false}$. Hence, $|\{j \mid \mathit{CrashedInCS}_j\}| = 0$ holds. According to the liveness property in Definition 6.2.2(b), every non-crashed process eventually enters its critical section.

Assume that in the above computation sequence every non-crashed process P_i is at location loc_3 trying to enter its critical section, i.e., $\mathsf{X_i} = \mathit{true}$ and $\mathsf{TRY_i} = \mathit{false}$ hold. In addition, assume that registers $\mathsf{ORD_i}$ record such values that function *choice* determines that every process crashed trying to enter the critical section has a higher priority than any non-crashed process. Consequently, those ℓ crashed processes are included in set A_i and $A_i \geq \ell$ holds, for every non-crashed process P_i, see Figure 4.3. Therefore, every non-crashed process P_i always sets its register $\mathsf{TRY_i}$ to *false* and never attempts to enter its critical section. Registers $\mathsf{ORD_i}$ then remain unchanged as well as process priorities.

In (Abraham, Dolev, Herman, and Koll 2001), a process is defined to be crashed if it stops executing its code or if it forever executes its critical section. A process forever executing its remainder section is, on the other hand, defined to be not crashed. In (Abraham, Dolev, Herman, and Koll 2001), the SLEX algorithm satisfies the following (weaker) liveness property: Every non-crashed process that is not forever in its remainder section eventually enters its critical section, provided less than ℓ processes are crashed. The latter liveness property is expressed by

$$(|\{j \mid \mathit{CrashedInCS}_j \vee \mathit{CrashedTrying}_j\}| < \ell \wedge \neg \mathit{crashed}_i) \rightarrow \Box\Diamond \mathit{InCS}_i \ .$$

CHAPTER 7

CORRECTNESS PROOF

In this chapter, we give a formal proof that the program in Chapter 4 meets the safety and liveness properties given in Definition 6.2.2. In Section 7.1, we formulate some basic properties of the program. Those properties are used throughout the rest of the correctness proof. Our proof of the safety property in Definition 6.2.2(a) is the subject of Section 7.2. Section 7.3 contains a proof of the liveness property in Definition 6.2.2(b).

7.1 Basic Properties

The semantics of read and write commands in Chapter 5 and the initial condition in Section 6.1 ensure that every process P_i's sub-history is sequential. Moreover, every read operation on a register is preceded by some write operation on that register. Hence, the following lemma holds.

Lemma 7.1.1. *In every state of any computation sequence of program $\mathcal{S}$, the history is well-formed.* ■

Since the critical section and the remainder section are modeled by single atomic transitions, see Chapter 5, a non-crashed process does not remain in its critical or its remainder section forever. Consequently, every non-crashed process eventually reaches location ℓ_1.

Lemma 7.1.2. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property holds:*

$$\Box\Diamond loc_i = \ell_1 \ . \qquad \blacksquare$$

Assume that, in some state of a computation of program $\mathcal{S}$, the last write operation on register X immediately precedes the last read operation by process P_i on X. Then the value of that last read operation on X equals the value of the last write operation on X. If register Y is an array, a similar property holds for every element of that array.

Lemma 7.1.3. *Let P_i be a non-crashed process, X be a register, Y be an array, and k be an index in that array. For all computation sequences of program $\mathcal{S}$, the following properties hold:*

a) $\Box((w = \mathit{LastWrt}(X) \wedge \mathit{LastRd}(r, i, X) \wedge w \sqsubset r) \rightarrow \mathit{Val}(r) = \mathit{Val}(w))$ *and*

b) $\Box((w = \mathit{LastWrt}(Y) \wedge \mathit{LastRd}(r, i, Y) \wedge w \sqsubset r) \rightarrow \mathit{Val}(r)[k] = \mathit{Val}(w)[k])$. $\blacksquare$

7.2 Safety Proof

In our proof, we identify all states of program $\mathcal{S}$ in which the safety property in Definition 6.2.2(a) is satisfied. Those states are identified by predicate Synch_i, for every process P_i. Predicate Synch_i holds if process P_i has read all the values recorded in registers $\mathsf{VEC_j}$, for every process P_j, and, thereafter, updated its register $\mathsf{VEC_i}$.

Definition 7.2.1. *Let P_i be a process. Predicate Synch_i is true if, for every process P_j, the following property holds:*

$$\exists r \cdot \mathit{OpType}(r) = \langle i, \mathsf{rd}, \mathsf{VEC_j} \rangle \wedge r \prec \mathit{LastWrt}(\mathsf{VEC_i}) \ .$$

We say that process P_i is eventually synchronized if Synch_i eventually holds in any computation sequence of program $\mathcal{S}$.

Once predicate Synch_i holds in some state of a computation sequence, then Synch_i continues to hold. Thus Synch_i is a stable property. Note that if P_i is crashed, the history does not contain any read operations performed by P_i. Therefore, P_i is never synchronized if it is crashed. In Section 7.3 we will show that every non-crashed process is eventually synchronized.

7.2.1 Properties of Registers $\mathsf{VEC_i}$

We first characterize the values of operations on registers $\mathsf{VEC_i}$. The first lemma expresses that the value read by process P_i on register $\mathsf{VEC_j}$ is recorded in variable $v_i[j]$, for every process P_j. In addition, process P_i can only complete a read operation on register $\mathsf{VEC_j}$, for any process P_j, if P_i is at location ℓ_5 or ℓ_{15}.

Lemma 7.2.1. *Let P_i be a non-crashed process and P_j be a (possibly crashed) process. For all computation sequences of program $\mathcal{S}$, the following properties hold:*

a) $\Box(\mathit{LastRd}(r, i, \mathsf{VEC_j}) \rightarrow v_i[j] = \mathit{Val}(r))$ *and*

b) $\Box(\mathit{LastRd}(r, i, \mathsf{VEC_j}) \rightarrow (\mathit{LastRd}(r, i, \mathsf{VEC_j})\, \mathcal{U}\, (\mathit{loc}_i = \ell_5 \vee \mathit{loc}_i = \ell_{15})))$. ■

The next lemma expresses that process P_i can perform a write operation on register $\mathsf{VEC_i}$ only if P_i is at location ℓ_{11}.

Lemma 7.2.2. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property holds:*

$$\Box w = \mathit{LastWrt}(\mathsf{VEC_i}) \bullet (w = \mathit{LastWrt}(\mathsf{VEC_i})\, \mathcal{W}\, \mathit{loc}_i = \ell_{11}) \ . \qquad ■$$

Next we give two properties relating the values of variables vec_i, try_i, and $\mathit{old\_try}_i$ to write operations on register $\mathsf{VEC_i}$. If process P_i is at location ℓ_{11}, then $\mathit{try}_i \wedge \neg \mathit{old\_try}_i$ holds. And if P_i is at ℓ_{12} and $\mathit{try}_i \wedge \neg \mathit{old\_try}_i$ holds, the value most recently written to $\mathsf{VEC_i}$ equals vec_i.

Lemma 7.2.3. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following properties eventually hold:*

a) $\Box(loc_i = \ell_{11} \rightarrow (try_i \wedge \neg old\_try_i))$ *, and*

b) $\Box((loc_i = \ell_{12} \wedge try_i \wedge \neg old\_try_i) \rightarrow vec_i = \mathbf{VEC}_{\mathsf{i}})$. ■

In the next lemma, we relate the value of variable vec_i to v_i. Process P_i eventually takes the transition from location ℓ_6 to ℓ_7. Thus, eventually P_i is at a location in $[\ell_7, \ell_{12}]$ and variable vec_i records the value computed by function $report(v_i, i)$.

Lemma 7.2.4. *Let P_i be a non-crashed process and P_j be a distinct (possibly crashed) process. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned}
&\Box(loc_i \in [\ell_7, \ell_{12}] \\
&\quad \rightarrow (vec_i[i] \neq v_i[i][i] \wedge vec_i[i] \neq v_i[j][i].first \wedge vec_i[i] \neq v_i[j][i].second \\
&\qquad \wedge vec_i[j].first = v_i[j][j] \wedge vec_i[j].second = v_i[i][j].first)) \ .
\end{aligned}$$
■

The following lemma characterizes the dependence of write operations on register $\mathsf{VEC_i}$ on the values previously read from registers $\mathsf{VEC_j}$. Because of Lemma 7.2.1(b), the value of register $\mathsf{VEC_j}$ can be read by non-crashed process P_i only if P_i is at location ℓ_5 or ℓ_{15}. Because of Lemma 7.2.1(a), that value is recorded in variable $v_i[j]$. Because of Lemma 7.2.2, the value of register $\mathsf{VEC_i}$ can be modified only if P_i is at location ℓ_{11}. Because of Lemma 7.2.3(a)(b), if P_i is at ℓ_{11}, the value of variable vec_i is written to $\mathsf{VEC_i}$. And because of Lemma 7.2.4, that value of vec_i is dependent on the values recorded in $v_i[j]$. Hence, the following lemma holds:

Lemma 7.2.5. *Let P_i be a non-crashed process and P_j be a distinct (possibly crashed) process. In addition, let w be a write operation on register* $\mathsf{VEC_i}$*, r be a read operation on register* $\mathsf{VEC_i}$ *by process P_i, and r' be a read operation on register* $\mathsf{VEC_j}$ *by process*

P_i. For all computation sequences of program $\mathcal{S}$, the following properties eventually hold:

a) $\Box(r \sqsubset w \rightarrow (\mathit{Val}(w)[i] \neq \mathit{Val}(r)[i] \wedge \mathit{Val}(w)[j].\mathit{second} = \mathit{Val}(r)[j].\mathit{first}))$ *and*

b) $$\Box(r' \sqsubset w \rightarrow (\mathit{Val}(w)[i] \neq \mathit{Val}(r')[i].\mathit{first} \wedge \mathit{Val}(w)[i] \neq \mathit{Val}(r')[i].\mathit{second} \\ \wedge \mathit{Val}(w)[j].\mathit{first} = \mathit{Val}(r')[j]))\ .$$ ■

Assume that in some computation sequence there exist two consecutive write operations on register $\mathsf{VEC_i}$ in the history, and some read operation by process P_i on register $\mathsf{VEC_i}$ immediately precedes the later of the two consecutive write operations on $\mathsf{VEC_i}$. Assume that some process P_j's read operation on $\mathsf{VEC_i}$ immediately precedes some write operation on register $\mathsf{VEC_j}$. If the first of the two consecutive write operations on $\mathsf{VEC_i}$ is a value source of the P_j's read operation on $\mathsf{VEC_i}$, then the following holds: (1) The color written to $\mathsf{VEC_i}[i]$ by the later of the two consecutive write operations on $\mathsf{VEC_i}$ differs from the color written in $\mathsf{VEC_j}[i].\mathit{first}$; and (2) the color written in $\mathsf{VEC_j}[j]$ differs from the color written to $\mathsf{VEC_i}[j].\mathit{second}$ by the later write operation on $\mathsf{VEC_i}$.

Lemma 7.2.6. *Let P_i and P_j be distinct non-crashed processes. Let w_i and w_i^1 be write operations on register $\mathsf{VEC_i}$, and w_j be a write operation on register $\mathsf{VEC_j}$. In addition, let r_i be a read operation on register $\mathsf{VEC_i}$ by process P_i, and r_j be a read operation on register $\mathsf{VEC_i}$ by process P_j. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box((w_i^1 \sqsubset r_i \sqsubset w_i \wedge r_j \sqsubset w_j \wedge \mathit{ValSrc}(r_j, w_i^1)) \\ \rightarrow (\mathit{Val}(w_i)[i] \neq \mathit{Val}(w_j)[i].\mathit{first} \wedge \mathit{Val}(w_j)[j] \neq \mathit{Val}(w_i)[j].\mathit{second}))\ .$$

Proof. Let P_i and P_j be distinct non-crashed processes. Consider an arbitrary computation sequence σ of program $\mathcal{S}$ such that the property in Lemma 7.2.5 always holds. Let w_i and w_i^1 be some write operations on register $\mathsf{VEC_i}$, w_j be some write operation on register $\mathsf{VEC_j}$, r_i be a read operation on $\mathsf{VEC_i}$ by P_i, and r_j be a read operation on $\mathsf{VEC_i}$ by P_j. Moreover, assume that $w_i^1 \sqsubset r_i \sqsubset w_i$ holds, $r_j \sqsubset w_j$ holds, and $\mathit{ValSrc}(r_j, w_i^1)$ holds in σ.

Because of Lemma 7.1.3(a) and from predicate *ValSrc*, it follows that $\mathit{Val}(r_j) = \mathit{Val}(w_i^1) = \mathit{Val}(r_i)$ holds in σ. We distinguish two cases:

a) $\mathit{Val}(w_i)[i] \neq \mathit{Val}(w_j)[i].\mathit{first}$ holds in σ.
Because of Lemma 7.2.5(a), $\mathit{Val}(w_i)[i] \neq \mathit{Val}(r_i)[i]$ is satisfied. And because of Lemma 7.2.5(b), $\mathit{Val}(w_j)[i].\mathit{first} = \mathit{Val}(r_j)[i]$ holds. Consequently, $\mathit{Val}(w_i)[i] \neq \mathit{Val}(w_j)[i].\mathit{first}$ holds.

b) $\mathit{Val}(w_j)[j] \neq \mathit{Val}(w_i)[j].\mathit{second}$ holds in σ.
Because of Lemma 7.2.5(b), $\mathit{Val}(w_j)[j] \neq \mathit{Val}(r_j)[j].\mathit{first}$ is satisfied. Because of Lemma 7.2.5(a), $\mathit{Val}(w_i)[j].\mathit{second} = \mathit{Val}(r_i)[j].\mathit{first}$ holds. Consequently, $\mathit{Val}(w_j)[j] \neq \mathit{Val}(w_i)[j].\mathit{second}$ holds. ■

Assume a read operation r_i on register $\mathsf{VEC_j}$ by process P_i immediately precedes some write operation w_i on register $\mathsf{VEC_i}$. Assume a read operation r_j on $\mathsf{VEC_i}$ by process P_j immediately precedes the last write operation w_j on $\mathsf{VEC_j}$ and w_j is immediately preceded by some other write operation on $\mathsf{VEC_j}$. In addition, assume that w_i^1 is the write operation on $\mathsf{VEC_i}$ immediately preceding w_i, and there exists some write operation w_i^2 on $\mathsf{VEC_i}$ preceding w_i^1. If w_i^2 is the value source of operation r_j, then the color written by operation w_i in $\mathsf{VEC_i}[i]$ differs from the color written by operation w_j in $\mathsf{VEC_j}[i].\mathit{second}$.

Lemma 7.2.7. *Let P_i and P_j be distinct non-crashed processes that are eventually synchronized. Let w_i, w_i^1, and w_i^2 be write operations on register $\mathsf{VEC_i}$, and w_j and w_j^1 be write operations on register $\mathsf{VEC_j}$. Let r_i be a read operation on $\mathsf{VEC_j}$ by P_i, and r_j be a read operation on $\mathsf{VEC_i}$ by P_j. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box((w_i^2 \prec w_i^1 \sqsubset r_i \sqsubset w_i \wedge w_j^1 \sqsubset r_j \sqsubset w_j \wedge w_j = \mathit{LastWrt}(\mathsf{VEC_j}) \wedge \mathit{ValSrc}(r_j, w_i^2)) \\ \rightarrow \mathit{Val}(w_i)[i] \neq \mathit{Val}(w_j)[i].\mathit{second}) \enspace .$$

Proof. Let P_i and P_j be distinct non-crashed processes. Consider an arbitrary computation sequence σ of program $\mathcal{S}$ such that processes P_i and P_j are synchronized and the properties in Lemmata 7.2.5 and 7.2.6 always hold. Assume that w_i, w_i^1, w_i^2, w_j, w_j^1, r_i, and r_j are the operation as described in the lemma and that $w_i^2 \prec w_i^1 \sqsubset r_i \sqsubset w_i$ holds, $w_j^1 \sqsubset r_j \sqsubset w_j$ holds, $w_j = \mathit{LastWrt}(\mathsf{VEC_j})$ holds, and $\mathit{ValSrc}(r_j, w_i^2)$ holds in σ.

First note that w_i^1 does not precede r_j. (Otherwise, by the program semantics, read operation r_j can read only the value written by w_i^1 or w_i.) Hence, $\mathit{Start}(r_j) < \mathit{Concl}(w_i^1)$ is satisfied, and because $w_j^1 \sqsubset r_j$ and $w_i^1 \sqsubset r_i$ hold, it follows that $w_j^1 \prec r_i$ is satisfied in σ. We therefore distinguish two cases:

a) $\mathit{ValSrc}(r_i, w_j^1)$ holds in σ.
   Then, because of Lemma 7.2.6, $\mathit{Val}(w_i)[i] \neq \mathit{Val}(w_j)[i].\mathit{second}$ holds.

b) $\mathit{ValSrc}(r_i, w_j)$ holds in σ.
   Then, because of Lemma 7.2.5(b), $\mathit{Val}(w_i)[i] \neq \mathit{Val}(w_j)[i].\mathit{second}$ holds. ■

Now we relate the values recorded in variables $v_i[i]$ and $v_i[j]$, provided that processes P_i and P_j are not crashed. Intuitively, if process P_i is at a location in $[\ell_{17}, \ell_{20}]$ and register $\mathsf{VEC_i}$ was updated later than register $\mathsf{VEC_j}$, the color recorded in variable $v_i[i][i]$ differs from the *first* or the *second* color recorded in $v_i[j][i]$.

Lemma 7.2.8. *Let P_i and P_j be distinct non-crashed processes that are eventually synchronized. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box((\mathit{loc}_i \in [\ell_{17}, \ell_{20}] \wedge \mathit{Concl}(\mathit{LastWrt}(\mathsf{VEC_j})) < \mathit{Concl}(\mathit{LastWrt}(\mathsf{VEC_i}))) \\ \rightarrow (v_i[i][i] \neq v_i[j][i].\mathit{first} \vee v_i[i][i] \neq v_i[j][i].\mathit{second})) \enspace .$$

Proof. Consider any state of a computation sequence of program $\mathcal{S}$ after processes P_i and P_j are synchronized and the property in Lemma 7.2.7 always holds. Then there exist read operations r' and r'' such that $\mathit{LastRd}(r', i, \mathsf{VEC_i})$ and $\mathit{LastRd}(r'', i, \mathsf{VEC_j})$ hold. Because of Lemma 7.2.1(a), $v_i[i] = \mathit{Val}(r')$ and $v_i[j] = \mathit{Val}(r'')$. Assume P_i is at a location in $[\ell_{17}, \ell_{20}]$ and $w_i = \mathit{LastWrt}(\mathsf{VEC_i})$, $w_j = \mathit{LastWrt}(\mathsf{VEC_j})$, and $\mathit{Concl}(w_j) < \mathit{Concl}(w_i)$ hold. Because of Lemmata 7.2.1(b) and 7.2.2, $w_i \sqsubset r'$ and $w_i \sqsubset r''$ is satisfied. Consequently, $\mathit{ValSrc}(r', w_i)$ holds because w_i is the last write operation to $\mathsf{VEC_i}$, and $\mathit{ValSrc}(r'', w_j)$ holds because w_j is the last write operation to $\mathsf{VEC_j}$ and $\mathit{Concl}(w_j) < \mathit{Concl}(w_i)$ holds.

Let r_i be P_i's read operation on register $\mathsf{VEC_i}$ such that $r_i \sqsubset w_i$. Let w_i^1 be the write operation on register $\mathsf{VEC_i}$ such that $w_i^1 \sqsubset r_i$, and w_i^2 be a write operation on $\mathsf{VEC_i}$ such that $w_i^2 \prec w_i^1$, provided such an operation exists. Let r_j be P_j's read operation on $\mathsf{VEC_i}$ such that $r_j \sqsubset w_j$. Let w_j^1 be the write operation to register $\mathsf{VEC_j}$ such that $w_j^1 \sqsubset r_j$, and w_j^2 be a write operation to $\mathsf{VEC_j}$ such that $w_j^2 \prec w_j^1$, provided such an operation exists. Because Synch_i and Synch_j hold, operations r_i, w_i^1, r_j, and w_j^1 exist. Since $r_j \sqsubset w_j$ and $\mathit{Concl}(w_j) < \mathit{Concl}(w_i)$ hold, we consider three cases:

a) $\mathit{ValSrc}(r_j, w_i)$ holds.

   Then $\mathit{Start}(w_i) < \mathit{Concl}(r_j)$ holds. Because $r_i \sqsubset w_i$ and $r_j \sqsubset w_j$, we consider two subcases:

   i) $\mathit{ValSrc}(r_i, w_j^1)$ holds.

      Then, because of Lemma 7.2.6, $\mathit{Val}(w_i)[i] \neq \mathit{Val}(w_j)[i].\mathit{first}$ holds. Consequently, $v_i[i] \neq v_i[j].\mathit{first}$ holds.

   ii) $\mathit{ValSrc}(r_i, w_j^2)$ holds, for some w_j^2.

      Then, because of Lemma 7.2.7, $\mathit{Val}(w_i)[i] \neq \mathit{Val}(w_j)[i].\mathit{second}$ holds. Consequently, $v_i[i] \neq v_i[j].\mathit{second}$ holds.

b) $\mathit{ValSrc}(r_j, w_i^1)$ holds.

Then, because of Lemma 7.2.6, $\mathit{Val}(w_i)[i] \neq \mathit{Val}(w_j)[i].\mathit{first}$ holds. Consequently, $v_i[i] \neq v_i[j].\mathit{first}$ holds.

c) $\mathit{ValSrc}(r_j, w_i^2)$ holds, for some w_i^2.

Then, because of Lemma 7.2.7, $\mathit{Val}(w_i)[i] \neq \mathit{Val}(w_j)[i].\mathit{second}$ holds. Consequently, $v_i[i] \neq v_i[j].\mathit{second}$ holds. ■

We now relate the values read from registers $\mathsf{VEC_i}$ and $\mathsf{VEC_j}$ and recorded in variables $v_i[i]$ and $v_i[j]$, provided that process P_j is crashed. Because of Lemma 7.2.5(b), if process P_j is crashed, the color written in $\mathsf{VEC_i}[i]$ differs from the *first* and *second* colors in $\mathsf{VEC_j}[i]$. Thereafter, the values of $\mathsf{VEC_i}$ and $\mathsf{VEC_j}$ are recorded in variables $v_i[i]$ and $v_i[j]$, respectively, when P_i is at a location in $[\ell_{17}, \ell_{20}]$.

Lemma 7.2.9. *Let P_i be a non-crashed process that is eventually synchronized, and P_j be a crashed process. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box(\mathit{loc}_i \in [\ell_{17}, \ell_{20}] \rightarrow (v_i[i][i] \neq v_i[j][i].\mathit{first} \wedge v_i[i][i] \neq v_i[j][i].\mathit{second})) \ . \qquad ■$$

7.2.2 Properties of Registers $\mathsf{TRY_i}$

In this subsection, we formulate properties about operations on registers $\mathsf{TRY_i}$. The first lemma expresses that the value read by process P_i on register $\mathsf{TRY_j}$ is recorded in variable $t_i[j]$, for every process P_j. Moreover, process P_i can complete a read operation on register $\mathsf{TRY_j}$, for any process P_j, only if P_i is at location $\ell_{8.3}$ or in ℓ_{16}.

Lemma 7.2.10. *Let P_i be a non-crashed process and P_j be a (possibly crashed) process. For all computation sequences of program $\mathcal{S}$, the following properties hold:*

a) $\Box(\mathit{LastRd}(r, i, \mathsf{TRY_j}) \rightarrow t_i[j] = \mathit{Val}(r))$ *and*

b) $\Box(\mathit{LastRd}(r, i, \mathsf{TRY_j}) \rightarrow (\mathit{LastRd}(r, i, \mathsf{TRY_j})\, \mathcal{U}\, (\mathit{loc}_i = \ell_{8.3} \vee \mathit{loc}_i = \ell_{16})))$. ■

We next express that process P_i can perform a write operation on register $\mathsf{TRY_i}$ only if it is at location ℓ_9 or ℓ_{20}.

Lemma 7.2.11. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property holds:*

$$\Box w = \mathit{LastWrt}(\mathsf{TRY_i}) \bullet (w = \mathit{LastWrt}(\mathsf{TRY_i})\ \mathcal{W}\ (\mathit{loc}_i = \ell_9 \vee \mathit{loc}_i = \ell_{20}))\ . \qquad \blacksquare$$

The value of variable $\mathit{old\_try}_i$ remains the same until process P_i is at location ℓ_8.

Lemma 7.2.12. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property holds:*

$$\Box y = \mathit{old\_try}_i \bullet (y = \mathit{old\_try}_i\ \mathcal{W}\ \mathit{loc}_i = \ell_8)\ . \qquad \blacksquare$$

The value of variable try_i remains the same until process P_i is at location ℓ_9.

Lemma 7.2.13. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property holds:*

$$\Box y = \mathit{try}_i \bullet (y = \mathit{try}_i\ \mathcal{W}\ \mathit{loc}_i = \ell_9)\ . \qquad \blacksquare$$

If process P_i is at a location in $[\ell_8, \ell_9]$, the value of variable $\mathit{old\_try}_i$ equals the value recorded in register $\mathsf{TRY_i}$.

Lemma 7.2.14. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box(\mathit{loc}_i \in [\ell_8, \ell_9] \rightarrow \mathit{old\_try}_i = \mathbf{TRY}_{\mathsf{i}})\ . \qquad \blacksquare$$

If process P_i is at a location in $[\ell_{10}, \ell_{20}]$, the value of variable try_i equals the value recorded in register $\mathsf{TRY_i}$. If P_i is at a location in $[\ell_{13}, \ell_{20}]$, the value of variable try_i is *true*.

Lemma 7.2.15. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following properties eventually hold:*

a) $\Box(loc_i \in [\ell_{10}, \ell_{20}] \rightarrow try_i = \mathbf{TRY}_\mathsf{i})$ *and*

b) $\Box(loc_i \in [\ell_{13}, \ell_{20}] \rightarrow try_i = true)$. ■

Once register $\mathsf{TRY_i}$ records *false*, that value can change only if process P_i is at location ℓ_9 and $try_i \wedge \neg old\_try_i$ holds.

Lemma 7.2.16. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box(\mathbf{TRY}_\mathsf{i} = false \rightarrow (\mathbf{TRY}_\mathsf{i} = false\, \mathcal{W}\, (loc_i = \ell_9 \wedge try_i \wedge \neg old\_try_i))) \ .$$

Proof. Consider any state of a computation sequence of program $\mathcal{S}$ after the property in Lemma 7.2.14 always holds. Assume that $\mathbf{TRY}_\mathsf{i} = false$ holds. Because of Lemma 7.2.11, a write operation on $\mathsf{TRY_i}$ can be performed only if P_i is at location ℓ_{20} or ℓ_9. If P_i is at ℓ_{20}, the value of $\mathsf{TRY_i}$ remains *false*. Assume process P_i is at location ℓ_9. Then, because of Lemmata 7.2.12 and 7.2.14, $\neg old\_try_i$ holds. If try_i also holds, we are done. If $\neg try_i$ holds at location ℓ_9, then the value of the last write operation to register $\mathsf{TRY_i}$ remains *false*. ■

Once register $\mathsf{TRY_i}$ records *true*, that value can be changed only if process P_i is at location ℓ_9 and $\neg try_i \wedge old\_try_i$ holds, or P_i is at ℓ_{20}.

Lemma 7.2.17. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box(\mathbf{TRY}_\mathsf{i} = true \rightarrow (\mathbf{TRY}_\mathsf{i} = true \\ \mathcal{W}\, ((loc_i = \ell_9 \wedge \neg try_i \wedge old\_try_i) \vee loc_i = \ell_{20}))) \ .$$

Proof. Consider any state of a computation sequence of program $\mathcal{S}$ after the property in Lemma 7.2.14 always holds. Assume that $\mathbf{TRY}_\mathsf{i} = true$. Because of Lemma 7.2.11, a write operation to $\mathsf{TRY_i}$ can be performed only if P_i is at location ℓ_{20} or ℓ_9. If P_i is at ℓ_{20}, we are done. Assume process P_i is at location ℓ_9. Then, because of Lemmata

7.2.12 and 7.2.14, $\mathit{old\_try}_i$ holds. If $\neg \mathit{try}_i$ also holds, we are done. Assume try_i holds at location ℓ_9. Then the value of the last write operation to register $\mathsf{TRY_i}$ remains *true*. ■

Assume processes P_i and P_j are synchronized and the last write operation on register $\mathsf{VEC_j}$ completes earlier than the last write operation on $\mathsf{VEC_i}$. If P_i is at a location in $[\ell_{17}, \ell_{20}]$, variable $t_i[j]$ is *true*.

Lemma 7.2.18. *Let P_i and P_j be non-crashed processes that are eventually synchronized. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned}&\Box((\mathit{loc}_i \in [\ell_{17}, \ell_{20}] \wedge \mathit{loc}_j \in [\ell_{17}, \ell_{20}] \\ &\quad \wedge \mathit{Concl}(\mathit{LastWrt}(\mathsf{VEC_j})) \leq \mathit{Concl}(\mathit{LastWrt}(\mathsf{VEC_i}))) \\ &\quad \to t_i[j] = \mathit{true}) \ .\end{aligned}$$

Proof. Consider any state of a computation sequence of program $\mathcal{S}$ after processes P_i and P_j are synchronized and the properties in Lemmata 7.2.3, 7.2.15, 7.2.16, and 7.2.17 always hold. Assume P_j is at a location in $[\ell_{17}, \ell_{20}]$ and $w_j = \mathit{LastWrt}(\mathsf{VEC_j})$ holds. Because P_j is synchronized, there exists write operation w' to register $\mathsf{TRY_j}$ such that $w' \sqsubset w_j$. Because of Lemmata 7.2.3 and 7.2.15(a), the value of w' is *true*. Because of Lemma 7.2.15(b), the value of the last write operation to $\mathsf{TRY_j}$ is also *true*. Because of Lemmata 7.2.3, 7.2.16, and 7.2.17, every write operation to $\mathsf{TRY_j}$ performed after w' has value *true*. Assume $\mathit{loc}_i \in [\ell_{17}, \ell_{20}]$, $w_i = \mathit{LastWrt}(\mathsf{VEC_i})$, and $\mathit{Concl}(w_j) \leq \mathit{Concl}(w_i)$ hold. Let r be the last read operation on $\mathsf{TRY_j}$ by P_i. Because P_i is at a location in $[\ell_{17}, \ell_{20}]$, $w_i \prec r$ holds. Because $w' \sqsubset w_j$ and $\mathit{Concl}(w_j) \leq \mathit{Concl}(w_i)$ hold, it follows that $w' \prec r$ is satisfied. Because of Lemma 7.1.3(a), we obtain that $\mathit{Val}(r) = \mathit{true}$. Consequently, because of Lemma 7.2.10(a), $t_i[j] = \mathit{true}$. ■

Finally, the value of the last write operation on register $\mathsf{TRY_j}$ remains unchanged if P_j is crashed. That value is recorded in variable $t_i[j]$, for every non-crashed process P_i.

Lemma 7.2.19. *Let P_j be a crashed process. For all computation sequences of program $\mathcal{S}$ and every non-crashed process P_i, the following property eventually holds:*

$$\Box t_i[j] = \mathbf{TRY}_{\mathsf{j}} \ . \qquad \blacksquare$$

7.2.3 The Safety Property

We first formulate some properties about the values of variable B_i.

Lemma 7.2.20. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following properties eventually hold:*

a) $\Box(loc_i \in [\ell_{18}, \ell_{20}] \rightarrow B_i = \{j \mid t_i[j] \wedge \neg dominates(v_i, j, i)\})$ *and*

b) $\Box(loc_i \in [\ell_{19}, \ell_{20}] \rightarrow |B_i| \leq \ell)$. $\blacksquare$

Next we show which processes are identified by a non-crashed process as trying to enter the critical section. Assume a non-crashed process P_i is synchronized and is at a location in $[\ell_{18}, \ell_{20}]$. Then variable B_i records every process P_k satisfying the following: Either P_k is crashed in the critical section, or the value of register $\mathsf{TRY_k}$ is *true* and the last write operation to $\mathsf{VEC_k}$ completed earlier than the last write operation to $\mathsf{VEC_i}$.

Lemma 7.2.21. *Let $T = \{k \mid Synch_k \wedge loc_k \in [\ell_{18}, \ell_{20}]\}$ be a set of processes and $\omega : \{1, \ldots, |T|\} \rightarrow T$ be the one-to-one correspondence satisfying the following: If $w_{\omega(j)}$ is the last write to register $\mathsf{VEC}_{\omega(\mathsf{j})}$ and $w_{\omega(j+1)}$ is the last write to register $\mathsf{VEC}_{\omega(\mathsf{j+1})}$, then $Concl(w_{\omega(j)}) < Concl(w_{\omega(j+1)})$ holds, for all $j = 1, \ldots, |T| - 1$. Then, for all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box(1 \leq i \leq |T| \rightarrow |B_{\omega(i)}| \geq i + |\{k \mid \mathit{CrashedInCS}_k\}|) \ .$$

Proof. Consider any state of a computation sequence of program $\mathcal{S}$ after the properties in Lemmata 7.2.9, 7.2.8, 7.2.19, 7.2.18, and 7.2.20 always hold. Let ω be the correspondence in the lemma. Assume $1 \leq i \leq |T|$ holds and P_j is an arbitrary process.

a) $\{k \mid \mathit{CrashedInCS}_k\} \subseteq B_{\omega(i)}$ holds.

Because $\omega(i) \in T$, $\mathit{Synch}_{\omega(i)}$ and $\mathit{loc}_{\omega(i)} \in [\ell_{18}, \ell_{20}]$ hold. Let P_j be a process in $\{k \mid \mathit{CrashedInCS}_k\}$. Because P_j is crashed in the critical section, $j \notin T$ and therefore $j \neq \omega(i)$. Moreover, because of Lemma 7.2.19, $t_{\omega(i)}[j]$ is *true*. Because of Lemma 7.2.9, $\neg \mathit{dominates}(v_{\omega(i)}, j, \omega(i))$ holds. Consequently, because of Lemma 7.2.20(a), $j \in B_{\omega(i)}$ holds.

b) $\{\omega(k) \mid 1 \leq k < i\} \subseteq B_{\omega(i)}$ holds.

Let P_j be a process in $\{\omega(k) \mid 1 \leq k < i\}$. Then, for some $k < i$, $j = \omega(k)$ holds. Because $\omega(k) \in T$ and $\omega(i) \in T$ hold, $\mathit{Synch}_j \wedge \mathit{loc}_j \in [\ell_{18}, \ell_{20}]$ and $\mathit{Synch}_{\omega(i)} \wedge \mathit{loc}_{\omega(i)} \in [\ell_{18}, \ell_{20}]$ hold. Moreover, the last write operation to register $\mathsf{VEC_j}$ completed earlier than the last write operation to register $\mathsf{VEC_{\omega(i)}}$. Therefore, because of Lemma 7.2.18, $t_{\omega(i)}[j]$ is *true*. Because of Lemma 7.2.8, predicate $\neg \mathit{dominates}(v_{\omega(i)}, j, \omega(i))$ holds. Consequently, because of Lemma 7.2.20(a), $P_j \in B_{\omega(i)}$ holds.

c) $\{\omega(i)\} \subseteq B_{\omega(i)}$ holds.

Let $j = \omega(i)$ hold. Because $j \in T$, Synch_j and $\mathit{loc}_j \in [\ell_{18}, \ell_{20}]$ hold. Because of Lemma 7.2.18, $t_j[j]$ is *true*. Also $\neg \mathit{dominates}(v_j, j, j)$ trivially holds. Consequently, because of Lemma 7.2.20(a), $P_j \in B_{\omega(i)}$ holds.

From the above it follows that $B_{\omega(i)} \supseteq \{\omega(k) \mid 1 \leq k \leq i\} \cup \{k \mid \mathit{CrashedInCS}_k\}$. Consequently, $|B_{\omega(i)}| \geq i + |\{k \mid \mathit{CrashedInCS}_k\}|$ holds. ■

The next theorem states that the self-stabilizing ℓ-exclusion algorithm satisfies the safety property.

Theorem 7.1. *For all computation sequences of program $\mathcal{S}$, the following property holds:*

$$|\{j \mid \mathit{CrashedInCS}_j\}| \leq \ell \rightarrow \Diamond\Box\, |\{i \mid \mathit{InCS}_i\}| \leq \ell \ .$$

Proof. Consider an arbitrary computation sequence σ of program $\mathcal{S}$. We first show that every non-crashed process P_i which enters and leaves its critical section infinitely often eventually remains synchronized. When P_i leaves the critical section, it writes *false* to register $\mathsf{TRY_i}$. By Lemma 7.1.2, P_i is eventually at location ℓ_5 and reads the values recorded in registers $\mathsf{VEC_j}$, for every process P_j. Because of Lemma 7.2.16, process P_i sets $\mathsf{TRY_i}$ to *true* and attempts to enter its critical section only if $loc_i = \ell_9 \wedge try_i \wedge \neg old\_try_i$ holds. Because of Lemmata 7.2.13, 7.2.12, and 7.2.3(b), thereafter, the value of vec_i is written in register $\mathsf{VEC_i}$; hence, Synch_i holds.

Consider any suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that the properties in Lemmata 7.2.20 and 7.2.21 always hold and, for every process P_i, either Synch_i always holds or $\neg\mathit{Synch}_i$ always holds. Note that if $\neg\mathit{Synch}_i$ always holds, process P_i remains outside the critical section or it is crashed.

Assume the number of processes crashed in the critical section is at most ℓ. Because of Lemma 7.2.20(b), process P_i is in the critical section only if $|B_i| < \ell$ holds. Because of Lemma 7.2.21, there is a linear ordering of non-crashed processes in the critical section such that every process P_i records in variable B_i at least the indices of all processes crashed in the critical section and processes with indices lesser in the above ordering than P_i. Hence, there are at most $\ell - |\{k \mid \mathit{CrashedInCS}_k\}|$ processes for which $|B_i| \leq \ell$ holds. Consequently, the number of processes in the critical section is at most ℓ. ∎

7.3 Liveness Proof

In this section, we give a formal proof that the SLEX algorithm in Chapter 4 satisfies the liveness property in Definition 6.2.2(b). In Subsection 7.3.1 we formulate some properties of operations on registers $\mathsf{X_i}$ and $\mathsf{ORD_i}$. Subsection 7.3.2 presents several properties about shifting the values recorded in registers $\mathsf{VEC_i}$. Thereafter, in Subsection 7.3.3, we show that every non-crashed process P_i eventually identifies the set of processes which are crashed or repeatedly enter and leave their critical sections while P_i is attempting to enter its critical section. In Subsection 7.3.4, we state several properties of functions *Select* and *Prior*. Those properties are used in Subsection 7.3.5, where we show that the number of processes that can enter the critical section before some specific process is bounded. In Subsection 7.3.6, we show that every non-crashed process eventually has the highest priority or it enters its critical section. Finally, in Subsection 7.3.7, we show that if at most ℓ processes are crashed in the critical section and some non-crashed process has the highest priority among non-crashed processes, then that highest priority process eventually enters its critical section.

7.3.1 Properties of Registers $\mathsf{X_i}$ and $\mathsf{ORD_i}$

A write operation on register $\mathsf{X_i}$ is performed only if process P_i is at location ℓ_2 or ℓ_{21}. If the value most recently written to $\mathsf{X_i}$ is *false*, that value remains *false* unless P_i is at ℓ_2. And if the value most recently written to $\mathsf{X_i}$ is *true*, $\mathsf{X_i}$'s value remains *true* unless P_i is at ℓ_{21}.

Lemma 7.3.1. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following properties hold:*

a) $\Box w = \mathit{LastWrt}(\mathsf{X_i}) \bullet (w = \mathit{LastWrt}(\mathsf{X_i})\ \mathcal{W}\ (loc_i = \ell_2 \vee loc_i = \ell_{21}))$ *,*

b) $\Box(\mathbf{X}_\mathsf{i} = \mathit{false} \rightarrow (\mathbf{X}_\mathsf{i} = \mathit{false}\ \mathcal{W}\ loc_i = \ell_2))$ *, and* *c)* $\Box(\mathbf{X}_\mathsf{i} = \mathit{true} \rightarrow (\mathbf{X}_\mathsf{i} = \mathit{true}\, \mathcal{W}\, \mathit{loc}_i = \ell_{21}))$. ■

The value of a read operation by process P_i on register X_k is recorded in variable $x_i[k]$, for every process P_k. Process P_i completes a read operation on X_k only if P_i is at location $\ell_{8.4}$ and program variable j_i equals k.

Lemma 7.3.2. *Let P_i and P_k be processes. For all computation sequences of program $\mathcal{S}$, the following properties eventually hold:*

a) $\Box(\mathit{LastRd}(r, i, \mathsf{X}_\mathsf{k}) \rightarrow x_i[k] = \mathit{Val}(r))$ *and*

b) $\Box(\mathit{LastRd}(r, i, \mathsf{X}_\mathsf{k}) \rightarrow (\mathit{LastRd}(r, i, \mathsf{X}_\mathsf{k})\, \mathcal{W}\, (\mathit{loc}_i = \ell_{8.4} \wedge j_i = k)))$. ■

A write operation on register ORD_i is performed only if process P_i is at location ℓ_{23}.

Lemma 7.3.3. *Let P_i be a process. For all computation sequences of program $\mathcal{S}$, the following property holds:*

$$\Box w = \mathit{LastWrt}(\mathsf{ORD}_\mathsf{i}) \bullet (w = \mathit{LastWrt}(\mathsf{ORD}_\mathsf{i})\, \mathcal{W}\, \mathit{loc}_i = \ell_{23}) \ . \qquad ■$$

The value of a read operation by process P_i on register ORD_k is recorded in variable $\mathit{ord}_i[k]$, for every process P_k. Eventually, process P_i completes a read operation on ORD_k only if P_i is at location $\ell_{8.7}$, program variable j_i equals k, and $a_i[k]$ equals *true*.

Lemma 7.3.4. *Let P_i and P_k be processes. For all computation sequences of program $\mathcal{S}$, the following properties eventually hold:*

a) $\Box(\mathit{LastRd}(r, i, \mathsf{ORD}_\mathsf{k}) \rightarrow \mathit{ord}_i[k] = \mathit{Val}(r))$ *and*

b) $\Box(\mathit{LastRd}(r, i, \mathsf{ORD}_\mathsf{k})$
$\rightarrow (\mathit{LastRd}(r, i, \mathsf{ORD}_\mathsf{k})\, \mathcal{W}\, (\mathit{loc}_i = \ell_{8.7} \wedge j_i = k \wedge a_i[k] = \mathit{true})))$. ■

The value of variable $a_i[k]$ does not change unless non-crashed process P_i is at location $\ell_{8.5}$ and $j_i = k$, for any process P_k. In addition, if P_i is at a location in $[\ell_{8.8}, \ell_{8.10}]$, then the value of $a_i[k]$ equals $x_i[k] \vee t_i[k]$.

Lemma 7.3.5. *Let P_i be a non-crashed process. For every process P_k and all computation sequences of program $\mathcal{S}$, the following properties eventually hold:*

a) $\Box y = a_i[k] \bullet (a_i[k] = y \,\mathcal{W}\, (loc_i = \ell_{8.5} \wedge j_i = k))$,

b) $\Box (loc_i \in [\ell_{8.8}, \ell_{8.10}] \rightarrow (a_i[k] = x_i[k] \vee t_i[k]))$, *and*

c) $\Box a_i[i] = true$. ■

The value of variable $ord_i[k]$ does not change unless non-crashed process P_i is at location $\ell_{8.7}$, $j_i = k$, and $a_i[k]$ equals *true*, for any process P_k.

Lemma 7.3.6. *Let P_i be a non-crashed process. For every process P_k and all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box y = ord_i[k] \bullet (y = ord_i[k] \,\mathcal{W}\, (loc_i = \ell_{8.7} \wedge j_i = k \wedge a_i[k] = true)) \ . \qquad ■$$

If some non-crashed process P_i is at location ℓ_{23}, then the value of program variable row_i equals $change(ord_i, \gamma(a_i), i)$.

Lemma 7.3.7. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box (loc_i = \ell_{23} \rightarrow row_i = change(ord_i, \gamma(a_i), i)) \ . \qquad ■$$

If the value most recently written to register $\mathsf{TRY_i}$ is *true*, then the value most recently written to register $\mathsf{X_i}$ is also *true*, for every non-crashed process P_i.

Lemma 7.3.8. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box (\mathbf{TRY}_{\mathsf{i}} = true \rightarrow \mathbf{X}_{\mathsf{i}} = true) \ . \qquad ■$$

Every non-crashed process eventually performs read operations on registers $\mathsf{VEC_i}$, $\mathsf{TRY_i}$, and $\mathsf{X_i}$, for every process P_i.

Lemma 7.3.9. *Let P_j be a non-crashed process and P_i be a (possibly crashed) process. For all computation sequences of program $\mathcal{S}$, the following properties hold:*

a) $\Box(w = \mathit{LastWrt}(\mathsf{VEC_i}) \bullet \Diamond\exists r \cdot \mathit{LastRd}(r, j, \mathsf{VEC_i}) \wedge w \prec r)$,

b) $\Box(w = \mathit{LastWrt}(\mathsf{TRY_i}) \bullet \Diamond\exists r \cdot \mathit{LastRd}(r, j, \mathsf{TRY_i}) \wedge w \prec r)$, *and*

c) $\Box(w = \mathit{LastWrt}(\mathsf{X_i}) \bullet \Diamond\exists r \cdot \mathit{LastRd}(r, j, \mathsf{X_i}) \wedge w \prec r)$. ■

7.3.2 Shifting Colors in Registers $\mathsf{VEC_i}$

Assume that no write operation is ever performed on register $\mathsf{VEC_i}$, for some process P_i. Then, for any distinct process P_j, either color $\mathsf{VEC_j}[i].\mathit{first}$ eventually remains set to the color of process P_i, or eventually register $\mathsf{VEC_j}$ is never updated.

Lemma 7.3.10. *Let P_i and P_j be distinct processes. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned}&\Box(w_i = \mathit{LastWrt}(\mathsf{VEC_i}) \bullet \Box w_i = \mathit{LastWrt}(\mathsf{VEC_i})\\ &\quad \rightarrow (\quad \Diamond\Box\mathbf{VEC}_{\mathsf{j}}[i].\mathit{first} = \mathbf{VEC}_{\mathsf{i}}[i]\\ &\qquad \vee \Diamond(w_j = \mathit{LastWrt}(\mathsf{VEC_j}) \bullet \Box w_j = \mathit{LastWrt}(\mathsf{VEC_j})))) \ .\end{aligned}$$

Proof. Let P_i and P_j be distinct processes. Consider an arbitrary computation sequence σ of program $\mathcal{S}$ after the property in Lemma 7.2.4 always holds. Assume that variable w_i is frozen to the last write operation on $\mathsf{VEC_i}$ and always $w_i = \mathit{LastWrt}(\mathsf{VEC_i})$ holds in σ. In addition assume that P_j is not crashed and always $w_j = \mathit{LastWrt}(\mathsf{VEC_j}) \bullet \Diamond w_j \neq \mathit{LastWrt}(\mathsf{VEC_j})$ holds in σ. (Otherwise, the property formulated in the lemma is trivially true.)

Because of Lemmata 7.1.3(a), 7.2.1(a), and 7.3.9(a), there exists suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that always $v_j[i] = \mathbf{VEC}_{\mathsf{i}}$ holds. Because of Lemma 7.2.4, it always holds in $\sigma^{(n)}$ that if $\mathit{loc}_j = \ell_{11}$ holds, then $\mathit{vec}_j[i].\mathit{first} = v_j[i][i] = \mathbf{VEC}_{\mathsf{i}}[i]$ holds (1). Because of Lemma 7.2.2, always $\Diamond \mathit{loc}_j = \ell_{11}$ holds. Hence, there exists suffix $\sigma^{(n_1)}$, $n_1 \geq n$, such that $\mathbf{VEC}_{\mathsf{j}} = vec_j$ holds in $\sigma^{(n_1)}$. And because of (1), always $\mathbf{VEC}_{\mathsf{j}}[i].\mathit{first} = \mathbf{VEC}_{\mathsf{i}}[i]$ holds in $\sigma^{(n_1)}$. ■

Assume that no write operation is ever performed on register $\mathsf{VEC_i}$, for some process P_i. Then, for any process P_j distinct from P_i, either colors $\mathsf{VEC_j}[i].\mathit{first}$ and $\mathsf{VEC_j}[i].\mathit{second}$ eventually remain set to the color of process P_i, or eventually register $\mathsf{VEC_j}$ is never updated.

Lemma 7.3.11. *Let P_i and P_j be distinct processes. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned} &\Box(w_i = \mathit{LastWrt}(\mathsf{VEC_i}) \bullet \Box w_i = \mathit{LastWrt}(\mathsf{VEC_i}) \\ &\quad \rightarrow (\quad \Diamond\Box(\mathbf{VEC}_{\mathsf{j}}[i].\mathit{second} = \mathbf{VEC}_{\mathsf{j}}[i].\mathit{first} = \mathbf{VEC}_{\mathsf{i}}[i]) \\ &\qquad \vee \Diamond(w_j = \mathit{LastWrt}(\mathsf{VEC_j}) \bullet \Box w_j = \mathit{LastWrt}(\mathsf{VEC_j})))) \ . \end{aligned}$$

Proof. Let P_i and P_j be distinct processes. Consider an arbitrary computation sequence σ of program $\mathcal{S}$ after the properties in Lemmata 7.2.4 and 7.3.10 always hold. Assume that variable w_i is frozen to the last write operation on $\mathsf{VEC_i}$ and always $w_i = \mathit{LastWrt}(\mathsf{VEC_i})$ holds in σ (1). In addition assume that P_j is not crashed and always $w_j = \mathit{LastWrt}(\mathsf{VEC_j}) \bullet \Diamond w_j \neq \mathit{LastWrt}(\mathsf{VEC_j})$ holds in σ (2). (Otherwise, the property formulated in the lemma is trivially true.)

Because of (1) and Lemmata 7.1.3(a), 7.2.1(a), and 7.3.9(a), there exists suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that always $v_j[i] = \mathbf{VEC}_{\mathsf{i}}$ holds (3). Because of (1), (2), and Lemma 7.3.10, there exists suffix $\sigma^{(n_1)}$, $n_1 \geq n$, such that always $\mathbf{VEC}_{\mathsf{j}}[i].\mathit{first} = \mathbf{VEC}_{\mathsf{i}}[i]$ holds (4). Because of (1), (4), and Lemmata 7.1.3(b), 7.2.1(a), and 7.3.9(a), there exists suffix $\sigma^{(n_2)}$, $n_2 \geq n_1$, such that always $v_j[j][i].\mathit{first} = \mathbf{VEC}_{\mathsf{j}}[i].\mathit{first}$ holds (5). Because of Lemma 7.2.4, it always holds in $\sigma^{(n_2)}$ that if $loc_j = \ell_{11}$ holds, then $vec_j[i].\mathit{first} = v_j[i][i] \wedge vec_j[i].\mathit{second} = v_j[j][i].\mathit{first}$ holds (6). Because of Lemma 7.2.2, always $\Diamond loc_j = \ell_{11}$ holds. Hence, there exists suffix $\sigma^{(n_3)}$,

$n_3 \geq n_2$, such that $\mathbf{VEC}_{\mathsf{j}} = vec_j$ holds in $\sigma^{(n_3)}$. And because of (3), (4), (5), and (6), it follows that $\mathbf{VEC}_{\mathsf{j}}[i].second = \mathbf{VEC}_{\mathsf{j}}[i].first = \mathbf{VEC}_{\mathsf{i}}[i]$ always holds in $\sigma^{(n_3)}$. ■

7.3.3 Identification of Possibly Crashed Processes

Let expression $E_m(i)$ denote the set of all process indices identified by non-crashed process P_m as trying to enter their critical sections and which do not dominate process P_i. Intuitively, those are the crashed (or 'very slow') processes trying to enter the critical section.

Definition 7.3.1. *Let P_m be a non-crashed processes and P_i be a (possibly crashed) process. We define $E_m(i) = \{j \mid t_m[j] \wedge \neg dominates(v_m, j, i)\}$.*

For non-crashed process P_i, sets A_i and B_i can be described by means of sets E_i and program variable p_i. Variable p_i records indices of processes which have higher priorities than P_i.

Lemma 7.3.12. *Let P_i be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following properties eventually hold:*

a) $\square(loc_i = \ell_{18} \rightarrow B_i = E_i(i))$,

b) $\square(loc_i \in [\ell_{8.9}, \ell_{8.10}] \rightarrow p_i = \pi(choice(ord_i, \gamma(a_i)), i)$ *, and*

c) $\square(loc_i = \ell_{8.10} \rightarrow A_i = \{j \mid t_i[j] \wedge j \in p_i\} \cup \bigcup_{i' \in p_i} (E_i(i') \setminus \{i\}))$. ■

If no write operation is ever performed on register $\mathsf{VEC_i}$, for some process P_i, then either every non-crashed process P_m eventually identifies process P_j as permanently dominating P_i, or eventually register $\mathsf{VEC_j}$ is never updated.

Lemma 7.3.13. *Let P_m be a non-crashed process, and P_i and P_j be (possibly crashed) processes. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned}&\Box(w_i = \mathit{LastWrt}(\mathsf{VEC_i}) \bullet \Box w_i = \mathit{LastWrt}(\mathsf{VEC_i})\\ &\quad \rightarrow (\quad \Diamond\Box \mathit{dominates}(v_m, j, i)\\ &\qquad\quad \vee \Diamond(w_j = \mathit{LastWrt}(\mathsf{VEC_j}) \bullet \Box w_j = \mathit{LastWrt}(\mathsf{VEC_j})))) \ .\end{aligned}$$

Proof. Let P_m be a non-crashed process, and P_i and P_j be distinct processes. (If $i = j$, the property in the lemma is trivially true.) Consider an arbitrary computation sequence σ of program $\mathcal{S}$ after the property in Lemma 7.3.11 always holds. Assume that variable w_i is frozen to the last write operation on $\mathsf{VEC_i}$ and always $w_i = \mathit{LastWrt}(\mathsf{VEC_i})$ holds in σ (1). In addition assume that P_i is distinct from P_j and always $w_j = \mathit{LastWrt}(\mathsf{VEC_j}) \bullet \Diamond w_j \neq \mathit{LastWrt}(\mathsf{VEC_j})$ holds in σ (2). (Otherwise, the property in the lemma is trivially true.)

Because of (1) and Lemmata 7.1.3(a), 7.2.1(a), and 7.3.9(a), there exists suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that always $v_m[i] = \mathbf{VEC}_\mathsf{i}$ holds (3). Because of (1), (2), and Lemma 7.3.11, there exists suffix $\sigma^{(n_1)}$, $n_1 \geq n$, such that always $\mathbf{VEC}_\mathsf{j}[i].\mathit{second} = \mathbf{VEC}_\mathsf{j}[i].\mathit{first} = \mathbf{VEC}_\mathsf{i}[i]$ holds (4). Because of (1), (4), and Lemmata 7.1.3(b), 7.2.1(a), and 7.3.9(a), there exists suffix $\sigma^{(n_2)}$, $n_2 \geq n_1$, such that always $v_m[j][i].\mathit{second} = \mathbf{VEC}_\mathsf{j}[i].\mathit{second}$ and $v_m[j][i].\mathit{first} = \mathbf{VEC}_\mathsf{j}[i].\mathit{first}$ hold (5). Because of (1), (3), (4), and (5), it follows that $v_m[j][i].\mathit{second} = v_m[j][i].\mathit{first} = v_m[i][i]$ always holds in $\sigma^{(n_2)}$. Hence, $\mathit{dominates}(v_m, j, i)$ always holds in $\sigma^{(n_2)}$. ■

Assume that no write operation is ever performed on register $\mathsf{VEC_i}$, for some process P_i. Then every non-crashed process P_m eventually identifies the set of process indices that never dominate P_i.

Lemma 7.3.14. *Let P_m be a non-crashed process, P_i be a (possibly crashed) process, and $D_m(i) = \{j \mid \neg \mathit{dominates}(v_m, j, i)\}$ be a set of process indices. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box(w = \mathit{LastWrt}(\mathsf{VEC_i}) \bullet \Box w = \mathit{LastWrt}(\mathsf{VEC_i}) \rightarrow \Diamond y = D_m(i) \bullet \Box y = D_m(i)) \ .$$

Proof. Let P_m be a non-crashed process, and P_i be a (possible crashed) process. Consider an arbitrary computation sequence σ of program $\mathcal{S}$ after the property in Lemma 7.3.13 always holds. Assume that variable w_i is frozen to the last write operation on $\mathsf{VEC_i}$ and always $w_i = \mathit{LastWrt}(\mathsf{VEC_i})$ holds in σ (1).

Because of (1) and Lemmata 7.1.3(a), 7.2.1(a), and 7.3.9(a), there exists suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that always $v_m[i] = \mathbf{VEC}_\mathsf{i}$ holds (2). Consider an arbitrary process P_j. We distinguish two cases:

a) $\Diamond(w_j = \mathit{LastWrt}(\mathsf{VEC_j}) \bullet \Box w_j = \mathit{LastWrt}(\mathsf{VEC_j}))$ holds in $\sigma^{(n)}$.

   There exists suffix $\sigma^{(n_1)}$, $n_1 \geq n$, such that always $w_j = \mathit{LastWrt}(\mathsf{VEC_j}) \bullet \Box w_j = \mathit{LastWrt}(\mathsf{VEC_j})$ holds (3). Because of (3) and Lemmata 7.1.3(a), 7.2.1(a), and 7.3.9(a), there exists suffix $\sigma^{(n_2)}$, $n_2 \geq n_1$, such that always $v_m[j] = \mathbf{VEC}_\mathsf{j}$ holds (4). Because of (2) and (4), it follows that *dominates*(v_m, j, i) in $\sigma^{(n_2)}$ iff always *dominates*(v_m, j, i) in $\sigma^{(n_2)}$.

b) $\Box(w_j = \mathit{LastWrt}(\mathsf{VEC_j}) \bullet \Diamond w_j \neq \mathit{LastWrt}(\mathsf{VEC_j}))$ holds in $\sigma^{(n)}$.

   Because of (1) and Lemma 7.3.13, there exists suffix $\sigma^{(n_3)}$, $n_3 \geq n$, such that always *dominates*(v_m, j, i) holds in $\sigma^{(n_3)}$. ∎

Assume that no write operation is ever performed on register $\mathsf{VEC_i}$, for some process P_i. Then eventually the value written in register $\mathsf{TRY_i}$ remains the same.

Lemma 7.3.15. *Let P_i be a process. For all computation sequences of program $\mathcal{S}$, the following property holds:*

$$\Box(w = \mathit{LastWrt}(\mathsf{VEC_i}) \bullet \Box w = \mathit{LastWrt}(\mathsf{VEC_i}) \\ \to \Diamond(y = \mathbf{TRY}_\mathsf{i} \bullet \Box y = \mathbf{TRY}_\mathsf{i})) \ .$$

Proof. Let P_i be a process. Consider an arbitrary computation sequence σ of program $\mathcal{S}$ after the property in Lemma 7.2.16 always holds. Assume that variable w_i is frozen to the last write operation on $\mathsf{VEC_i}$ and always $w_i = \mathit{LastWrt}(\mathsf{VEC_i})$ holds in σ (1).

We prove the property in the lemma by contradiction. Thus assume that always $y = \mathbf{TRY}_{\mathsf{i}} \bullet \Diamond y \neq \mathbf{TRY}_{\mathsf{i}}$ holds in σ (2).

Because of (2), process P_i is not crashed and there exists suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that $\mathbf{TRY}_{\mathsf{i}} = \mathit{false}$ holds. Because of (2) and Lemma 7.2.16, there exists suffix $\sigma^{(n_1)}$, $n_1 \geq n$, such that $loc_i = \ell_9 \land try_i \land \neg old\_try_i$ holds. Consequently, eventually $loc_i = \ell_{11}$ and eventually $w_i \neq \mathit{LastWrt}(\mathsf{VEC_i})$ holds, which is a contradiction to (1). ∎

Finally assume that no write operation is ever performed on register $\mathsf{VEC_i}$, for some process P_i. Then every non-crashed process P_m eventually identifies the set of processes that are permanently attempting to enter their critical section and non-dominating to process P_i.

Lemma 7.3.16. *Let P_m be a non-crashed process. For all computation sequences of program $\mathcal{S}$, and every process P_i, the following property holds:*

$$\Box(w = \mathit{LastWrt}(\mathsf{VEC_i}) \bullet \Box w = \mathit{LastWrt}(\mathsf{VEC_i}) \\ \rightarrow \Diamond(y = E_m(i) \bullet \Box y = E_m(i))) \ .$$

Proof. Let P_m be a non-crashed process, and P_i be a (possible crashed) process. Consider an arbitrary computation sequence σ of program $\mathcal{S}$ after the properties in Lemmata 7.3.14 and 7.3.15 always hold. Assume that variable w_i is frozen to the last write operation on $\mathsf{VEC_i}$ and always $w_i = \mathit{LastWrt}(\mathsf{VEC_i})$ holds in σ (1).

Because of (1) and Lemma 7.3.14, there exists suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that $U = \{j \mid \neg \mathit{dominates}(v_m, j, i)\} \bullet \Box\, U = \{j \mid \neg \mathit{dominates}(v_m, j, i)\}$ holds. Because of (1) and Lemma 7.3.15, there exists suffix $\sigma^{(n_1)}$, $n_1 \geq n$, such that $V = \{j \mid \mathsf{TRY_j} = \mathit{true}\} \bullet \Box\, V = \{j \mid \mathsf{TRY_j} = \mathit{true}\}$ holds (2). Because of (2) and Lemmata 7.1.3(b), 7.2.10(a), and 7.3.9(a), there exists suffix $\sigma^{(n_2)}$, $n_2 \geq n_1$, such that always $t_m[j] = \mathbf{TRY}_{\mathsf{j}}$ holds, for every $j \in V$. Hence $W = \{j \mid t_m[j] = \mathit{true}\} \bullet \Box\, W =$

$\{j \mid t_m[j] = true\}$ holds in $\sigma^{(n_2)}$. By definition $E_m(i) = U \cap W$ holds. Therefore, $y = E_m(i) \bullet \Box y = E_m(i)$ holds in $\sigma^{(n_2)}$. ■

7.3.4 Properties of Functions *Select* and *Prior*

In this subsection, we describe several properties of functions *Select* and *Prior*. As in (Abraham, Dolev, Herman, and Koll 2001), we define the relation of matrix extension with respect to a set of indices. Intuitively, one matrix is an extension of another matrix if both the matrices record the same values in the rows in the set of indices. The concept of matrix extension is used to relate the values recorded in bit-matrices *ord* in the program.

Definition 7.3.2. *Let M and M' be two $r \times s$ matrices and A be a set of indices. We say that M' is an extension of M with respect to A, denoted by $M' \rhd_A M$, if row $M'[i]$ equals row $M[i]$, for every index $i \in A$.*

The relation of matrix extension with respect to a set of indices is an equivalence relation.

We next formulate three properties of function *Select*. Assume M is an $N \times N$ bit-matrix recording in row z, for every process P_z, the value of register $\mathsf{ORD_z}$. In addition, assume that k, $1 \leq k \leq N$, is an integer, and A is a set of process indices. (1) If the value $Prior(M, A, i, k)$ equals $M[i][k]$, then the value of function $Select(M, A, k)$ does not depend on the presence or absence of index i in set A. (2) In this case, process index i is not selected by function $Select(M, A, k)$. (3) The process index selected in column k of bit-matrix M does not depend on the values recorded in those rows of bit-matrix M which correspond to the process indices not present in set A.

Lemma 7.3.17. *Let P_i and P_j be processes, M be a bit-matrix, k, $1 \leq k \leq N$, be an integer, and A be a set of process indices such that $A \setminus \{i\} \neq \emptyset$ holds. Then the following properties hold:*

a) $M[i][k] = \mathit{Prior}(M, A, i, k) \rightarrow \mathit{Select}(M, A, k) = \mathit{Select}(M, A \setminus \{i\}, k)$ *,*

b) $\mathit{Select}(M, A, k) = \mathit{Select}(M, A \setminus \{i\}, k) \rightarrow \mathit{Select}(M, A, k) \neq i$ *, and*

c) $M' \rhd_A M \rightarrow \mathit{Select}(M, A, k) = \mathit{Select}(M', A, k)$ *.* ■

Intuitively, property (a) holds because if in column k the parity of process P_i is the same as the parity of its previous active process, by definition the parity of P_i does not influence the outcome of function *Select*; then also (b) P_i is not selected in column k. Moreover, (c) parities of inactive processes do not influence the outcome of function *Select*.

We next focus on function *Prior*. Using the same notation as above, we show that (1) The value of $\mathit{Prior}(M, A, i, k)$ does not depend on the presence of index i in set A. (2) The value of $\mathit{Prior}(M, A, i, k)$ does not depend on the presence of those indices j in set A which are different from the previous process of process P_i, or for which the value recorded in $M[j][k]$ equals $\mathit{Prior}(M, A, j, k)$. (3) The value of $\mathit{Prior}(M, A, i, k)$ does not depend on the values recorded in those rows of bit-matrix M which correspond to the process indices not present in set A.

Lemma 7.3.18. *Let P_i and P_j be processes, M be a bit-matrix, k, $1 \leq k \leq N$, be an integer, and A be a set of process indices such that $A \setminus \{i\} \neq \emptyset$ holds. Then the following properties hold:*

a) $\mathit{Prior}(M, A, i, k) = \mathit{Prior}(M, A \setminus \{i\}, i, k)$ *,*

b) $(j \neq \mathit{Prev}(A, i) \vee M[j][k] = \mathit{Prior}(M, A, j, k))$
$\rightarrow \mathit{Prior}(M, A, i, k) = \mathit{Prior}(M, A \setminus \{j\}, i, k)$*, and*

c) $M' \rhd_A M \rightarrow \mathit{Prior}(M, A, i, k) = \mathit{Prior}(M', A, i, k)$ *.* ■

In essence, property (a) holds because, by definition, the parity of the previous active process to process P_i (and not of P_i itself) is the new parity of P_i. Moreover, (b) if P_i is not the previous active process to process P_j, then the parity of P_i does not influence the new parity of P_j; similarly, if the parity of P_i is the same as the parity of its previous active process, the same parity will be assigned to P_j independently of whether P_i is active. Finally, (c) parities of inactive processes do not influence the new parity of P_i.

7.3.5 Bounding Changes of Register $\mathsf{ORD_i}[k]$

Intuitively, a flip on a register is defined as a (possibly incomplete) write operation that changes the value recorded in that register.

Definition 7.3.3. *Let P_i be a process, X be a register, Y be a register recording an array, k be an index in that array, and op be an operation. We define state predicates*

a) $$\mathit{Flip}(op, X) \equiv \mathit{OpType}(op) = \langle \mathsf{wrt}, X\rangle \wedge ((\mathit{OpType}(op') = \langle \mathsf{wrt}, X\rangle \wedge op' \sqsubset op) \rightarrow \mathit{Val}(op') \neq \mathit{Val}(op)) \ ,$$

b) $$\mathit{Flip}(op, Y[k]) \equiv \mathit{OpType}(op) = \langle \mathsf{wrt}, Y\rangle \wedge ((\mathit{OpType}(op') = \langle \mathsf{wrt}, Y\rangle \wedge op' \sqsubset op) \rightarrow \mathit{Val}(op')[k] \neq \mathit{Val}(op)[k]) \ ,$$

c) $op = \mathit{LastFlip}(X)$ *iff* $\mathit{Flip}(op, X) \wedge (\mathit{Flip}(op', X) \rightarrow op' \preceq op)$ *, and*

d) $op = \mathit{LastFlip}(Y[k])$ *iff* $\mathit{Flip}(op, Y[k]) \wedge (\mathit{Flip}(op', Y[k]) \rightarrow op' \preceq op)$ *.*

We say that operation op is a flip on register X (or register $Y[k]$) if $\mathit{Flip}(op, X)$ (or $\mathit{Flip}(op, Y[k])$) holds. The latest such operation op in the history is called the last flip on register X (or register $Y[k]$).

Note that the initial condition ensures that all histories of program $\mathcal{S}$ are well-formed. By definition, the first write operation on a register is a flip on that register. Therefore, the last flip on every register is well-defined and unique.

We later need a rank function to count the number of flips performed on register $\mathsf{ORD_i}[k]$, for any process P_i and index k in array $\mathsf{ORD_i}$.

Definition 7.3.4. *Let P_i be a process and k, $1 \leq k \leq N$, be an integer. We define function on states*

$$\mathit{FlipCount}(\mathsf{ORD_i}[k]) = |\{op \mid \neg \mathit{crashed}_i \wedge \mathit{Flip}(op, \mathsf{ORD_i}[k])\}| \ .$$

We next focus on properties of function *FlipCount*. Hereafter, we assume that P_i is a non-crashed process and k, $1 \leq k \leq N$, is an integer. (1) Since events are appended, but never removed from the history, the number of flips on register $\mathsf{ORD_i}[k]$ never decreases. (2) By Lemma 7.3.3, a flip operation on register $\mathsf{ORD_i}[k]$ can occur only when process P_i is at location ℓ_{23}. Thereafter, P_i eventually reaches location ℓ_1. Thus the number of flips on register $\mathsf{ORD_i}[k]$ increases by at most one before process P_i reaches location ℓ_1. For the same reason, (3) the number of flips on register $\mathsf{ORD_i}[k]$ remains the same unless process P_i reaches location ℓ_{23} and the value of $\mathsf{ORD_i}[k]$ differs from $\mathit{change}(\mathit{ord}_i, \gamma(a_i), i)[k]$.

Lemma 7.3.19. *Let P_i be a non-crashed process and k, $1 \leq k \leq N$, be an integer. For every computation sequence of program $\mathcal{S}$, the following properties eventually hold:*

a) $\Box(f = \mathit{FlipCount}(\mathsf{ORD_i}[k]) \bullet \Box \mathit{FlipCount}(\mathsf{ORD_i}[k]) \geq f)$,

b) $\Box(f = \mathit{FlipCount}(\mathsf{ORD_i}[k]) \bullet \Diamond(\quad \mathit{loc}_i = \ell_1$
$\qquad\qquad \wedge\ \mathit{FlipCount}(\mathsf{ORD_j}[k]) \leq f + 1))$, *and*

c) $\Box(f = \mathit{FlipCount}(\mathsf{ORD_i}[k]) \bullet \mathit{FlipCount}(\mathsf{ORD_i}[k]) = f$
$\qquad \mathcal{W}\ (\quad \mathit{change}(\mathit{ord}_i, \gamma(a_i), i)[k] \neq \mathbf{ORD}_\mathsf{i}[k]$
$\qquad\qquad \wedge\ \mathit{loc}_i = \ell_{23}))$. ■

We now determine an upper bound of the number of flips performed on any register $\mathsf{ORD_j}[k]$ before non-crashed process P_j reads the values recorded in registers $\mathsf{TRY_i}$,

$\mathsf{X_i}$, and $\mathsf{ORD_i}$, or P_j determines that process P_i is not active. We assume that the values in registers $\mathsf{TRY_i}$, $\mathsf{X_i}$, and $\mathsf{ORD_i}$ remain the same. Otherwise, process P_i enters and leaves its critical section and there is nothing to prove.

Lemma 7.3.20. *Let P_i be a (possibly crashed) process, P_j be a non-crashed process, and c and k, $1 \leq c, k \leq N$, be integers. For every computation sequence of program $\mathcal{S}$, the following properties eventually hold:*

a) $$\Box(op = \mathit{LastFlip}(\mathsf{TRY_i}) \bullet \Box(op = \mathit{LastFlip}(\mathsf{TRY_i}) \wedge \mathit{Complete}(op)) \\ \rightarrow \mathit{FlipCount}(\mathsf{ORD_j}[k]) = f \bullet \Diamond(\quad loc_j = \ell_{8.8} \wedge \mathit{FlipCount}(\mathsf{ORD_j}[k]) \leq f+1 \\ \wedge \Box t_j[i] = \mathbf{TRY}_{\mathsf{i}})) \ ,$$

b) $$\Box(op = \mathit{LastFlip}(\mathsf{X_i}) \bullet \Box(op = \mathit{LastFlip}(\mathsf{X_i}) \wedge \mathit{Complete}(op)) \\ \rightarrow \mathit{FlipCount}(\mathsf{ORD_j}[k]) = f \bullet \Diamond(\quad loc_j = \ell_{8.8} \wedge \mathit{FlipCount}(\mathsf{ORD_j}[k]) \leq f+1 \\ \wedge \Box x_j[i] = \mathbf{X}_{\mathsf{i}})) \ , \text{ and}$$

c) $$\Box(op = \mathit{LastFlip}(\mathsf{ORD_i}[c]) \bullet \Box(op = \mathit{LastFlip}(\mathsf{ORD_i}[c]) \wedge \mathit{Complete}(op)) \\ \rightarrow \mathit{FlipCount}(\mathsf{ORD_j}[k]) = f \bullet \Diamond(\quad loc_j = \ell_{8.8} \wedge \mathit{FlipCount}(\mathsf{ORD_j}[k]) \leq f+1 \\ \wedge \Box(\quad loc_j \in [\ell_{8.5}, \ell_{8.7}] \vee \neg a_j[i] \\ \vee \mathit{ord}_j[i][c] = \mathbf{ORD}_{\mathsf{i}}[c]))) \ . \qquad \blacksquare$$

As intuitively described in Chapter 4, process P_i is active if the value recorded either in register $\mathsf{TRY_i}$ or in register $\mathsf{X_i}$ equals *true*. In addition, the last write operation modifying the value recorded in $\mathsf{TRY_i}$ or $\mathsf{X_i}$ has to be complete.

Definition 7.3.5. *Let P_i be a (possibly crashed) process. We define*

$$\mathit{Active}(i) \equiv \quad (\mathit{Complete}(\mathit{LastFlip}(\mathsf{TRY_i})) \wedge \mathbf{TRY}_{\mathsf{i}} = \mathit{true}) \\ \vee (\mathit{Complete}(\mathit{LastFlip}(\mathsf{X_i})) \wedge \mathbf{X}_{\mathsf{i}} = \mathit{true}) \ .$$

We say that P_i is active in column k of bit-matrix ord_j, for some non-crashed process P_j, if P_i is active and $a_j[i] \wedge 1 \leq k \leq |\gamma(a_j)| \wedge i \notin \rho(\mathit{choice}(ord_j, a_j), k)$ holds.

Intuitively, a process is active in a column if it is active and not selected in an earlier column.

For every non-crashed process P_i, the following properties hold: (1) If P_i is active, then P_i is at a location in $[\ell_1, \ell_{21}]$. (2) If P_i is active, then the last flip on register $\mathsf{X_i}$ is complete and the value recorded in $\mathsf{X_i}$ equals *true*. (3) If P_i is active, then the last write operation on register $\mathsf{ORD_i}$ is complete.

Lemma 7.3.21. *Let P_i be a non-crashed process. For every computation sequence of program $\mathcal{S}$, the following properties eventually hold:*

a) $\Box(\mathit{Active}(i) \rightarrow \mathit{loc}_i \in [\ell_1, \ell_{21}])$,

b) $\Box(\mathit{Active}(i) \rightarrow (\mathit{Complete}(\mathit{LastFlip}(\mathsf{X_i})) \wedge \mathbf{X}_{\mathsf{i}} = \mathit{true}))$, *and*

c) $\Box(\mathit{Active}(i) \rightarrow \mathit{Complete}(\mathit{LastWrt}(\mathsf{ORD_i})))$. ■

If non-crashed process P_i is active at a location in $[\ell_1, \ell_{18}]$, then either P_i remains active at a location in $[\ell_1, \ell_{18}]$, or P_i eventually enters its critical section.

Lemma 7.3.22. *Let P_i be a non-crashed process. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned}&\Box((\mathit{Active}(i) \wedge \mathit{loc}_i \in [\ell_1, \ell_{18}])\\ &\quad \rightarrow (\Box(\mathit{Active}(i) \wedge \mathit{loc}_i \in [\ell_1, \ell_{18}]) \vee \Diamond \mathit{InCS}_i))\ .\end{aligned}$$ ■

In our proof, we need to identify the states in which process P_i is changing the value recorded in register $\mathsf{ORD_i}[k]$. A new value (different from the previously written value) can be written to $\mathsf{ORD_i}[k]$ only if process P_i is not crashed. In addition, any write operation on $\mathsf{ORD_i}$ is performed only when P_i is not active. Finally, the new value of $\mathsf{ORD_i}[k]$ differs from the previously written value only if it is not the case that $\mathit{ord}_i[i][k] = \mathsf{ORD_i}[k] = \mathit{change}(\mathit{ord}_i, \gamma(a_i), i)[k]$.

Definition 7.3.6. *Let P_i be a process and k, $1 \leq k \leq N$, be an integer. We define*

$$\begin{aligned} \mathit{Flipping}(\mathsf{ORD_i}[k]) \equiv \quad & \neg \mathit{crashed}_i \wedge \neg \mathit{Active}(i) \\ \wedge (\quad & \mathit{ord}_i[i][k] \neq \mathbf{ORD}_\mathsf{i}[k] \\ & \vee \mathbf{ORD}_\mathsf{i}[k] \neq \mathit{change}(\mathit{ord}_i, \gamma(a_i), i)[k]) \ . \end{aligned}$$

We say that register $\mathsf{ORD_i}[k]$ *is flipping if* $\mathit{Flipping}(\mathsf{ORD_i}[k])$ *holds.*

If register $\mathsf{ORD_i}[k]$ is not flipping, then the last flip operation on that register is complete. In addition, the value recorded in $\mathsf{ORD_i}[k]$ remains unchanged. Consequently, if $\mathsf{ORD_i}[k]$ never flips, eventually every non-crashed process records the value of $\mathsf{ORD_i}[k]$ in $\mathit{ord}_j[i][k]$ or $a_j[i]$ is always *false*.

Lemma 7.3.23. *Let P_i be a (possibly crashed) process, P_j be a non-crashed process, and k, $1 \leq k \leq N$, be an integer. For every computation sequence of program $\mathcal{S}$, the following properties eventually hold:*

a) $\Box(\neg \mathit{Flipping}(\mathsf{ORD_i}[k]) \rightarrow \mathit{Complete}(\mathit{LastFlip}(\mathsf{ORD_i}[k])))$,

b) $\Box(y = \mathbf{ORD}_\mathsf{i}[k] \bullet (y = \mathbf{ORD}_\mathsf{i}[k] \ \mathcal{W} \ \mathit{Flipping}(\mathsf{ORD_i}[k])))$ *, and*

c) $\Box(\Box\neg \mathit{Flipping}(\mathsf{ORD_i}[k]) \rightarrow \Diamond(\Box\neg a_j[i] \vee \Box \mathit{ord}_j[i][c] = \mathbf{ORD}_\mathsf{i}[c]))$. ■

We next define three predicates characterizing sets of registers $\mathsf{ORD_i}[k]$ that are not flipping. Intuitively, column k of bit-matrix ORD is frozen if no register $\mathsf{ORD_z}[k]$ is flipping, for any process P_z. Columns below column k of bit-matrix ORD are frozen if every column preceding column k is frozen. Finally, registers below register $\mathsf{ORD_i}[k]$ are frozen (with respect to active process P_m) if columns below column k are frozen and no register $\mathsf{ORD_z}[k]$ is flipping such that the distance of process P_z from P_m is less than the distance of process P_i from P_m. Recall that the distance of process P_m from P_i is defined as $\mathit{dist}(P_m, P_i) = (i - m) \ \mathit{mod} \ N$. Later we will show that the number of register $\mathsf{ORD_i}[k]$'s flips is bounded provided that registers below $\mathsf{ORD_i}[k]$ are frozen.

Definition 7.3.7. *Let P_m and P_i be processes and k, $1 \leq k \leq N$, and c, $1 \leq c \leq N+1$, be integers. We define*

a) $\mathit{FrozenColumn}(k) \equiv \neg \mathit{Flipping}(\mathsf{ORD_z}[k])$,

b) $\mathit{FrozenColsBelow}(c) \equiv 1 \leq k < c \rightarrow \mathit{FrozenColumn}(k)$, *and*

c) $$\begin{aligned} \mathit{FrozenBelow}(\mathsf{ORD_i}[k], P_m) \equiv \quad & \mathit{Active}(m) \wedge \mathit{FrozenColsBelow}(k) \\ & \wedge (\mathit{dist}(P_m, P_z) < \mathit{dist}(P_m, P_i) \\ & \quad \rightarrow \neg \mathit{Flipping}(\mathsf{ORD_z}[k])) \ . \end{aligned}$$

We say that column k is frozen if $\mathit{FrozenColumn}(k)$ holds. We say that columns below column k are frozen if $\mathit{FrozenColsBelow}(k)$ holds. We say that registers below register $\mathsf{ORD_i}[k]$ are frozen (with respect to process P_m) if $\mathit{FrozenBelow}(\mathsf{ORD_i}[k], P_m)$ holds.

The next lemma, in essence, claims that the value $\mathit{change}(\mathit{ord}_j, \gamma(a_j), i)[k]$ remains unchanged provided some values in column k of bit-matrix ord_j preceding row i are unchanged. More precisely, for any process P_i and column k, the following property holds: If (1) process P_j is outside locations $[\ell_{8.5}, \ell_{8.7}]$; (2) the value of $\mathit{change}(\mathit{ord}_j, \gamma(a_j), i)[k]$ equals T; (3) process P_m (distinct from P_i) is always active in column k of bit-matrix ord_j and the value recorded in $\mathit{ord}_j[m][k]$ equals $\mathsf{ORD_m}[k]$; (4) every process P_z with the distance from P_m less than the distance of process P_i from P_m is either (4a) always active in column k of ord_j and the value recorded in $\mathit{ord}_j[z][k]$ equals $\mathsf{ORD_z}[k]$, or (4b) always if process P_j is outside locations $[\ell_{8.5}, \ell_{8.7}]$, then P_z is inactive in column k of ord_j or the value recorded in $\mathit{ord}_j[z][k]$ equals $\mathit{change}(\mathit{ord}_j, \gamma(a_j), z)[k]$; then the value of $\mathit{change}(\mathit{ord}_j, \gamma(a_j), i)[k]$ remains unchanged whenever P_j is outside locations $[\ell_{8.5}, \ell_{8.7}]$.

Lemma 7.3.24. *Let P_i and P_m be distinct (possibly crashed) processes, P_j be a non-crashed process, and k, $1 \leq k \leq N$, be an integer. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$
\begin{aligned}
\Box((\quad & loc_j \notin [\ell_{8.5}, \ell_{8.7}] \wedge change(ord_j, \gamma(a_j), i)[k] = T \\
& \wedge \Box(loc_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k \leq |\gamma(a_j)| \wedge m \notin \rho(choice(ord_j, \gamma(a_j)), k)) \\
& \wedge \forall P_z \cdot dist(P_m, P_z) < dist(P_m, P_i) \rightarrow \\
& \qquad (\quad \Box(\quad a_j[z] \wedge ord_j[z][k] = \mathbf{ORD_z}[k] \\
& \qquad\qquad\quad \wedge (loc_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow z \notin \rho(choice(ord_j, \gamma(a_j)), k))) \\
& \qquad\quad \vee \Box(\quad loc_j \in [\ell_{8.5}, \ell_{8.7}] \vee \neg a_j[z] \\
& \qquad\qquad\quad \vee ord_j[z][k] = change(ord_j, \gamma(a_j), z)[k] \\
& \qquad\qquad\quad \vee z \in \rho(choice(ord_j, \gamma(a_j)), k)))) \\
& \rightarrow \Box(loc_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow change(ord_j, \gamma(a_j), i)[k] = T)) \quad .
\end{aligned}
$$

Proof. Let P_i and P_m be distinct (possibly crashed) processes, P_j be a non-crashed process, and k, $1 \leq k \leq N$, be an integer. Consider an arbitrary computation sequence σ of program $\mathcal{S}$ such that (1) $loc_j \notin [\ell_{8.5}, \ell_{8.7}]$, (2) $change(ord_j, \gamma(a_j), i)[k] = Prior(ord_j, \gamma(a_j) \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k) = T$ holds, (3) $\Box(loc_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k \leq |\gamma(a_j)| \wedge m \notin \rho(choice(ord_j, \gamma(a_j)), k)))$ holds, and (4) for every process P_z such that $dist(P_m, P_z) < dist(P_m, P_i)$ either (4a) $\Box(a_j[z] \wedge ord_j[z][k] = \mathbf{ORD_z}[k] \wedge (loc_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow z \notin \rho(choice(ord_j, \gamma(a_j)), k)))$ holds or (4b) $\Box(loc_j \in [\ell_{8.5}, \ell_{8.7}] \vee \neg a_j[z] \vee ord_j[z][k] = change(ord_j, \gamma(a_j), z)[k] \vee z \in \rho(choice(ord_j, \gamma(a_j)), k))$ holds.

Let Z denote the set $\{z \mid dist(P_m, P_z) < dist(P_m, P_i) \wedge \Box((loc_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k \leq |\gamma(a_j)| \wedge z \notin \rho(choice(ord_j, \gamma(a_j)), k))) \wedge a_j[z] \wedge ord_j[z][k] = \mathbf{ORD_z}[k]\}$ in σ. Obviously, $m \in Z$ holds. Let $\sigma^{(n)}$, $n \geq 0$, be any suffix of computation sequence σ such that $loc_j \notin [\ell_{8.5}, \ell_{8.7}]$ holds. We show that $Prior(ord_j, \gamma(a_j) \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k) = Prior(ord_j, Z \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k)$ holds in $\sigma^{(n)}$. Let P_z be a process such that $z \notin Z$ holds. We distinguish two cases:

a) $dist(P_m, P_z) \geq dist(P_m, P_i)$ holds.

   Because $m \in Z$ holds in $\sigma^{(n)}$, it follows that process P_m is active in column k of bit-matrix ord_j. Since P_i is distinct from P_m and $dist(P_m, P_z) \geq dist(P_m, P_i)$, it follows that $z \neq Prev(\gamma(a_j) \setminus \rho(choice(ord_j, \gamma(a_j)), k), i)$ holds in $\sigma^{(n)}$. And because of Lemma 7.3.18(c)(d), $Prior(ord_j, \gamma(a_j) \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k) = Prior(ord_j, \gamma(a_j) \setminus \{z\} \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k)$ holds in $\sigma^{(n)}$.

b) $dist(P_m, P_z) < dist(P_m, P_i)$ holds.

Because $z \notin Z$, property (4b) holds for P_z in $\sigma^{(n)}$. Since $loc_j \notin [\ell_{8.5}, \ell_{8.7}]$ holds in $\sigma^{(n)}$, we distinguish the following three cases:

b1) $\neg a_j[z]$ holds in $\sigma^{(n)}$.

It follows that $z \notin \gamma(a_j)$ holds in $\sigma^{(n)}$. Consequently, by definition, $Prior(ord_j, \gamma(a_j) \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k) = Prior(ord_j, \gamma(a_j) \setminus \{z\} \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k)$ holds in $\sigma^{(n)}$.

b2) $ord_j[z][k] = change(ord_j, \gamma(a_j), z)[k]$ holds in $\sigma^{(n)}$.

$Prior(ord_j, \gamma(a_j) \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k) = Prior(ord_j, \gamma(a_j) \setminus \{z\} \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k)$ holds in $\sigma^{(n)}$ because of Lemma 7.3.18(c)(d).

b3) $z \in \rho(choice(ord_j, \gamma(a_j)), k)))$ holds in $\sigma^{(n)}$.

Thus $Prior(ord_j, \gamma(a_j) \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k) = Prior(ord_j, \gamma(a_j) \setminus \{z\} \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k)$ holds in $\sigma^{(n)}$.

Because of properties (a), (b1), (b2), and (b3), it follows that $Prior(ord_j, \gamma(a_j) \setminus \rho(choice(ord_j, \gamma(a_j)), k), i, k) = Prior(ord_j, Z, i, k)$ holds in $\sigma^{(n)}$. Hence, we conclude that $change(ord_j, \gamma(a_j), i)[k] = T$ holds $\sigma^{(n)}$. ■

Below we need the notion of an array (sequence) prefix.

Definition 7.3.8. *Let r be an array (sequence) of length $|r|$ and k, $1 \leq k \leq |r|+1$, be an integer. We define the k^{th} prefix of r, denoted by $\mu(r, k)$, as the array (sequence) s of length $k-1$ such that $s[i] = r[i]$ holds, for every index i, $1 \leq i < k$.*

The following lemma intuitively expresses that the process priorities assigned in the first $k-1$ columns of bit-matrix ord_j remain unchanged if values in those columns remain unchanged. More precisely, for any non-crashed process P_j and column k, the following property holds: If (1) process P_j is outside locations $[\ell_{8.5}, \ell_{8.7}]$; (2) the value of the k^{th} prefix of $\mathit{choice}(\mathit{ord}_j, \gamma(a_j))$ equals T; and, for every process P_i, (3a) P_i is always active and the k^{th} prefix of $\mathit{ord}_j[i]$ equals the k^{th} prefix of the value recorded in register $\mathsf{ORD_i}$, or (3b) always P_i is inactive or the k^{th} prefix of $\mathit{ord}_j[i]$ equals the k^{th} prefix of $\mathit{change}(\mathit{ord}_j, \gamma(a_j), i)$; then the value of the k^{th} prefix of $\mathit{choice}(\mathit{ord}_j, \gamma(a_j))$ remains unchanged when P_j is outside locations $[\ell_{8.5}, \ell_{8.7}]$.

Lemma 7.3.25. *Let P_j be a non-crashed process and k, $1 \le k \le N+1$, be an integer. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$
\begin{aligned}
\Box((\quad & \mathit{loc}_j \notin [\ell_{8.5}, \ell_{8.7}] \wedge k \le |\gamma(a_j)| + 1 \wedge \mu(\mathit{choice}(\mathit{ord}_j, \gamma(a_j)), k) = T \\
& \wedge \forall P_i \cdot \quad \Box(a_j[i] \wedge \mu(\mathit{ord}_j[i], k) = \mu(\mathbf{ORD}_{\mathsf{i}}, k)) \\
& \qquad\qquad \vee \Box(\quad \mathit{loc}_j \in [\ell_{8.5}, \ell_{8.7}] \vee \neg a_j[i] \\
& \qquad\qquad\qquad \vee \mu(\mathit{ord}_j[i], k) = \mu(\mathit{change}(\mathit{ord}_j, \gamma(a_j), i), k))) \\
& \rightarrow \Box(\mathit{loc}_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k \le |\gamma(a_j)| + 1 \\
& \qquad\qquad\qquad \wedge \mu(\mathit{choice}(\mathit{ord}_j, \gamma(a_j)), k) = T))) \ .
\end{aligned}
$$

Proof. Let P_j be a non-crashed process and k, $1 \le k \le N+1$, be an integer. We prove the property in the lemma by induction on k.

i) Assume $k = 1$.

   By definition, $k \le |\gamma(a_j)| + 1$ and $\mu(\mathit{choice}(\mathit{ord}_j, \gamma(a_j)), k) = \langle\rangle$ hold, and the property is trivially true.

ii) Assume that the property in the lemma holds for $k \ge 1$. We next show that the property holds for $k+1 \le N+1$.

   Consider an arbitrary computation sequence σ of program $\mathcal{S}$ such that (1) $\mathit{loc}_j \notin [\ell_{8.5}, \ell_{8.7}]$ holds, (2) $k+1 \le |\gamma(a_j)|+1$ holds, (3) $\mu(\mathit{choice}(\mathit{ord}_j, \gamma(a_j)), k+1) = T$ holds, and, for every process P_i, either (4a) $\Box(a_j[i] \wedge \mu(\mathit{ord}_j[i], k+1) =$

$\mu(\mathbf{ORD}_{\mathsf{i}}, k+1))$ holds, or (4b) $\Box(loc_j \in [\ell_{8.5}, \ell_{8.7}] \vee \neg a_j[i] \vee \mu(ord_j[i], k+1) = \mu(change(ord_j, \gamma(a_j), i), k+1))$ holds. In addition, assume that the inductive hypothesis (5) $\Box(loc_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k \leq |\gamma(a_j)|+1 \wedge \mu(choice(ord_j, \gamma(a_j)), k) = \mu(T, k)))$ holds in σ.

Let Z denote the set $\{i \mid \Box(a_j[i] \wedge \mu(ord_j[i], k+1) = \mu(\mathbf{ORD}_{\mathsf{i}}, k+1))\}$. Because of properties (1), (4b), (5), and Lemma 7.3.17(a)(b), it follows that $T[k+1] \in Z$ and (6) $|Z| \geq k+1$ always holds in σ. Let $\sigma^{(n)}$, $n \geq 0$, be any suffix of computation sequence σ such that $loc_j \notin [\ell_{8.5}, \ell_{8.7}]$ holds. We show that $Select(ord_j, \gamma(a_j) \setminus \rho(T, k+1), k+1) = Select(ord_j, Z \setminus \rho(T, k+1), k+1)$ holds in $\sigma^{(n)}$. Let P_i be a process such that $i \notin Z$ holds. Then property (4b) holds for P_i. Since $loc_j \notin [\ell_{8.5}, \ell_{8.7}]$ holds in $\sigma^{(n)}$, we distinguish two cases:

a) $\neg a_j[i]$ holds in $\sigma^{(n)}$.

   It follows that $i \notin \gamma(a_j)$ holds in $\sigma^{(n)}$. Consequently, $Select(ord_j, \gamma(a_j) \setminus \rho(T, k+1), k+1) = Select(ord_j, \gamma(a_j) \setminus \{i\} \setminus \rho(T, k+1), k+1)$ holds in $\sigma^{(n)}$.

b) $\mu(ord_j[i], k) = \mu(change(ord_j, \gamma(a_j), i), k)$ holds in $\sigma^{(n)}$.

   $Select(ord_j, \gamma(a_j) \setminus \rho(T, k+1), k+1) = Select(ord_j, \gamma(a_j) \setminus \{i\} \setminus \rho(T, k+1), k+1)$ holds in $\sigma^{(n)}$ because of Lemma 7.3.17(a).

Because of properties (a) and (b), it follows that $Select(ord_j, \gamma(a_j) \setminus \rho(T, k+1), k+1) = Select(ord_j, Z \setminus \rho(T, k+1), k+1)$ holds in $\sigma^{(n)}$. Hence, because of (6) and the inductive hypothesis, $k+1 \leq |\gamma(a_j)|+1 \wedge \mu(choice(ord_j, \gamma(a_j)), k+1) = T$ holds in $\sigma^{(n)}$. ■

We next show that if registers are frozen below register $\mathsf{ORD}_{\mathsf{m}}[k]$ with respect to process P_m, then within one flip on register $\mathsf{ORD}_{\mathsf{j}}[c]$ the values in columns preceding column k of bit-matrix ord_j remain unchanged. More precisely, for every process P_i, either (1) P_i is always active and the k^{th} prefix of $ord_j[i]$ equals the k^{th} prefix of the value recorded in register $\mathsf{ORD_i}$, or (2) always P_i is at a location in $[\ell_{8.5}, \ell_{8.7}]$ or P_i is inactive or the k^{th} prefix of $ord_j[i]$ equals the k^{th} prefix of $change(ord_j, \gamma(a_j), i)$.

Lemma 7.3.26. *Let P_i and P_m be (possibly crashed) processes, P_j be a non-crashed process, c, $1 \leq c \leq N$, and k, $1 \leq k \leq N+1$ be integers. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned}
&\Box((\Box \mathit{Active}(m) \wedge \Box \mathit{FrozenColsBelow}(k) \wedge \mathit{FlipCount}(\mathsf{ORD_j}[c]) = f) \\
&\quad \rightarrow \Diamond(\quad loc_j = \ell_{8.8} \wedge \mathit{FlipCount}(\mathsf{ORD_j}[c]) \leq f+1 \\
&\qquad\qquad \wedge (\quad \Box(\mathit{Active}(i) \wedge a_j[i] \wedge ord_j[i] = \mathbf{ORD}_{\mathsf{i}}) \\
&\qquad\qquad\quad \vee \Box(\neg \mathit{Active}(i) \wedge \neg a_j[i]) \\
&\qquad\qquad\quad \vee \Box(loc_j \notin [\ell_{8.5}, \ell_{8.7}] \\
&\qquad\qquad\qquad\qquad \rightarrow (\quad k \leq |\gamma(a_j)| + 1 \wedge m \notin \rho(choice(ord_j, \gamma(a_j)), k-1) \\
&\qquad\qquad\qquad\qquad\quad \wedge \mu(\mathbf{ORD}_{\mathsf{i}}, k) = \mu(change(ord_j, \gamma(a_j), i), k))) \\
&\qquad\qquad\quad \vee \Box(\quad loc_j \in [\ell_{8.5}, \ell_{8.7}] \vee k > |\gamma(a_j)| + 1 \\
&\qquad\qquad\qquad \vee m \in \rho(choice(ord_j, \gamma(a_j)), k-1))))) \ .
\end{aligned}$$

Proof. Let P_i and P_m be (possibly crashed) processes, P_j be a non-crashed process, c, $1 \leq c \leq N$, and k, $1 \leq k \leq N+1$ be integers. We prove the property in the lemma by induction on k.

i) Assume $k = 1$.

By definition, $\mu(\mathbf{ORD}_{\mathsf{i}}, k) = \langle\rangle = \mu(change(ord_j, \gamma(a_j), i), k)$ holds and the property is trivially true.

ii) Assume the property in the lemma holds for $k \geq 1$. We show that the property holds for $k+1 \leq N+1$.

We prove this case by induction on the distance of processes P_m and P_i.

a) Assume $dist(P_m, P_i) = 0$; that is, $P_i = P_m$.

By (1), (2) and Lemma 7.3.20(a)(b)(c), there exists suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that $\Box(\mathit{Active}(i) \wedge a_j[i] \wedge \mathit{ord}_j[i] = \mathbf{ORD}_{\mathsf{i}})$ holds.

b) Assume the property holds for every process P_z, $\mathit{dist}(P_m, P_z) = d$, $d \geq 0$. Let P_i be the process such that $\mathit{dist}(P_m, P_i) = d + 1 < N$. We show that the property holds for P_i.

Consider an arbitrary computation sequence σ of program $\mathcal{S}$ such that the properties in Lemmata 7.3.5, 7.3.21, and 7.3.20 always hold, $\Box\mathit{Active}(i) \vee \Box\neg\mathit{Active}(i) \vee \Box(\Diamond\mathit{Active}(i) \wedge \Diamond\neg\mathit{Active}(i))$ holds, and (1) $\Box\mathit{Active}(m)$ holds, (2) $\Box\mathit{FrozenColsBelow}(k{+}1)$ holds, and (3) $\mathit{FlipCount}(\mathsf{ORD_j}[c]) = f$ holds.

b1) Always $\mathit{Active}(i)$ holds in σ.

By (2) and Lemma 7.3.20(a)(b)(c), there exists suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that $\Box(\mathit{Active}(i) \wedge a_j[i] \wedge \mathit{ord}_j[i] = \mathbf{ORD}_{\mathsf{i}})$ holds.

b2) Always $\neg\mathit{Active}(i)$ holds in σ.

By Lemma 7.3.20(a)(b), there exists suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that $\Box(\neg\mathit{Active}(i) \wedge \neg a_j[i])$ holds.

b3) Always $\Diamond\mathit{Active}(i) \wedge \Diamond\neg\mathit{Active}(i)$ holds in σ.

By the inductive hypothesis in (ii), for every non-crashed process $P_{j'}$ and every process $P_{i'}$, there exists suffix $\sigma^{(n)}$, $n \geq 0$, of computation sequence σ such that (4) $\mathit{loc}_{j'} = \ell_{8.8} \wedge \mathit{FlipCount}(\mathsf{ORD_{j'}}[c]) \leq f + 1$ holds, and either (5a) $\Box(\mathit{Active}(i') \wedge a_{j'}[i'] \wedge \mathit{ord}_{j'}[i'] = \mathbf{ORD}_{\mathsf{i'}})$ holds or (5b) $\Box(\neg\mathit{Active}(i') \wedge \neg a_{j'}[i'])$ holds or (5c) $\Box(\mathit{loc}_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k \leq |\gamma(a_{j'})| + 1 \wedge m \notin \rho(\mathit{choice}(\mathit{ord}_{j'}, \gamma(a_{j'})), k - 1) \wedge \mu(\mathbf{ORD}_{\mathsf{i'}}, k) = \mu(\mathit{change}(\mathit{ord}_{j'}, \gamma(a_{j'}), i'), k)))$ holds or (5d) $\Box(\mathit{loc}_j \in [\ell_{8.5}, \ell_{8.7}] \vee k > |\gamma(a_{j'})| + 1 \vee m \in \rho(\mathit{choice}(\mathit{ord}_{j'}, \gamma(a_{j'})), k - 1))$ holds. If (5d) holds in $\sigma^{(n)}$, then (b) is trivially true. Therefore, assume (5d) does not hold in $\sigma^{(n)}$.

Because of (2) and Lemmata 7.3.20(c) and 7.3.23(b), it follows that (6) $\Box(loc_{j'} \in [\ell_{8.5}, \ell_{8.7}] \vee \neg a_{j'}[i'] \vee \mu(ord_{j'}[i'], k+1) = \mu(\mathbf{ORD}_{\mathsf{i}'}, k+1))$ holds in $\sigma^{(n)}$. Consequently, either (7a) $\Box(a_{j'}[i'] \wedge \mu(ord_{j'}[i'], k) = \mu(\mathbf{ORD}_{\mathsf{i}'}, k))$ or (7b) $\Box(loc_{j'} \in [\ell_{8.5}, \ell_{8.7}] \vee \neg a_{j'}[i'] \vee \mu(ord_{j'}[i'], k) = \mu(\mathbf{ORD}_{\mathsf{i}'}, k) = \mu(change(ord_{j'}, \gamma(a_{j'}), i'), k))$ holds in $\sigma^{(n)}$. Moreover, (8) $\Box(a_{j'}[m] \wedge ord_{j'}[m] = \mathbf{ORD}_{\mathsf{m}})$ holds in $\sigma^{(n)}$.

Consider the case that $k > |\gamma(a_{j'})| + 1$ holds in $\sigma^{(n)}$. Because of (8), there exists integer k', $1 \leq k' \leq |\gamma(a_{j'})| + 1$, such that $m \in \rho(choice(ord_{j'}, \gamma(a_{j'})), k')$ holds in $\sigma^{(n)}$. By Lemma 7.3.25, $\Box(loc_{j'} \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k' \leq |\gamma(a_{j'}) + 1| \wedge m \in \rho(choice(ord_{j'}, \gamma(a_{j'})), k')))$ holds in $\sigma^{(n)}$. Consequently, (5d) holds in $\sigma^{(n)}$, which is a contradiction. Therefore, assume that $k \leq |\gamma(a_{j'})| + 1$ holds in $\sigma^{(n)}$. It follows by Lemma 7.3.25 that (9) $T = \mu(choice(ord_{j'}, \gamma(a_{j'})), k) \bullet \Box(loc_{j'} \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k \leq |\gamma(a_{j'}) + 1| \wedge T = \mu(choice(ord_{j'}, \gamma(a_{j'})), k)))$ holds in $\sigma^{(n)}$. If $m \in \rho(T, k)$ holds in $\sigma^{(n)}$, then (b) is trivially true. Therefore, assume (10) $m \notin \rho(T, k)$ holds in $\sigma^{(n)}$.

By the inductive hypothesis in (b), for every process P_z, $dist(P_m, P_z) < dist(P_m, P_i)$, either (11a) $\Box(Active(z) \wedge a_{j'}[z] \wedge ord_{j'}[z] = \mathbf{ORD}_{\mathsf{z}})$ holds or (11b) $\Box(\neg Active(z) \wedge \neg a_{j'}[z])$ holds or (11c) $\Box(loc_{j'} \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k+1 \leq |\gamma(a_{j'})| + 1 \wedge m \in \rho(choice(ord_{j'}, \gamma(a_{j'})), k) \wedge \mu(\mathbf{ORD}_{\mathsf{z}}, k+1) = \mu(change(ord_{j'}, \gamma(a_{j'}), z), k+1)))$ or (11d) $\Box(loc_{j'} \in [\ell_{8.5}, \ell_{8.7}] \vee k+1 > |\gamma(a_{j'})| + 1 \vee m \in \rho(choice(ord_{j'}, \gamma(a_{j'})), k))$ holds in $\sigma^{(n)}$. If (11d) holds in $\sigma^{(n)}$, then again (b) is trivially true. Therefore, assume (11d) does not hold in $\sigma^{(n)}$.

Because of (6) and (9), either (12a) $\Box(a_{j'}[z] \wedge ord_{j'}[z][k] = \mathbf{ORD}_{\mathsf{z}}[k] \wedge (loc_{j'} \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow z \notin \rho(choice(ord_{j'}, \gamma(a_{j'})), k)))$ holds or (12b)

$\Box(loc_{j'} \notin [\ell_{8.5}, \ell_{8.7}] \vee \neg a_{j'}[z] \vee ord_{j'}[z][k] = change(ord_{j'}, \gamma(a_{j'}), z)[k] \vee z \in \rho(choice(ord_{j'}, \gamma(a_{j'})), k))$ holds in $\sigma^{(n)}$. Because of (9) and (10), $\Box(loc_{j'} \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k \leq |\gamma(a_{j'})| \wedge m \notin \rho(choice(ord_{j'}, \gamma(a_{j'})), k)))$ holds in $\sigma^{(n)}$. Consequently, because of Lemma 7.3.24, it follows that (13) $y = change(ord_{j'}, \gamma(a_{j'}), i)[k] \bullet \Box(loc_{j'} \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow y = change(ord_{j'}, \gamma(a_{j'}), i)[k])$ holds in $\sigma^{(n)}$.

Because of (b3), process P_i is not crashed. Let A denote set $\{i \mid \Box Active(i)\}$ in σ. Because of Lemma 7.3.20(a)(b)(c), eventually always bit-matrix ord_j is an extension of bit-matrix ord_i with respect to A and ord_i is an extension of ord_j with respect to A. Hence, because of Lemma 7.3.17(a)(c), eventually $\Box((loc_i \notin [\ell_{8.5}, \ell_{8.7}] \wedge loc_j \notin [\ell_{8.5}, \ell_{8.7}]) \rightarrow \mu(choice(ord_i, \gamma(a_i)), k) = \mu(choice(ord_j, \gamma(a_j)), k))$ is satisfied in σ. Then, because of Lemma 7.3.18(b)(c), eventually (14) $\Box((loc_i \notin [\ell_{8.5}, \ell_{8.7}] \wedge loc_j \notin [\ell_{8.5}, \ell_{8.7}]) \rightarrow \mu(change(ord_i, \gamma(a_i), i), k + 1) = \mu(change(ord_j, \gamma(a_j), i), k + 1)$ holds in σ.

Because of (b3), property (5c) holds for P_i. Because of (2) and (13), $\Box(loc_i \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow \mu(\mathbf{ORD}_{\mathsf{i}}, k{+}1) = \mu(change(ord_i, \gamma(a_i), i), k{+}1))$ holds in σ. Consequently, because of (9), (10) and (14), $\Box(loc_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k + 1 \leq |\gamma(a_j)| + 1 \wedge m \notin \rho(choice(ord_j, \gamma(a_j)), k) \wedge \mu(\mathbf{ORD}_{\mathsf{i}}, k + 1) = \mu(change(ord_j, \gamma(a_j), i), k + 1)))$ holds in $\sigma^{(n)}$. ■

In the following lemma, we claim that if registers are frozen below register $\mathsf{ORD}_\mathsf{j}[k]$ with respect to process P_m, then within one flip on register $\mathsf{ORD}_\mathsf{j}[c]$ the values below $\mathsf{ORD}_\mathsf{j}[k]$ of bit-matrix ord_j remain unchanged. More precisely, for every process P_i, either (1) process P_m remains inactive in column k of ord_j, or (2) for every process P_i such that the distance of P_i from P_m is less than the distance of P_j from P_m, (2a) P_i is always active in column k of ord_j and the k^{th} prefix of $ord_j[i]$ equals the k^{th} prefix of the value recorded in register ORD_i, or (2b) always P_i is inactive in column k of

ord_j or the k^{th} prefix of $ord_j[i]$ equals the k^{th} prefix of $change(ord_j, \gamma(a_j), i)$ or P_j is at a location in $[\ell_{8.5}, \ell_{8.7}]$.

Lemma 7.3.27. *Let P_j be a non-crashed process, P_m be a (possibly crashed) process, and k, $1 \leq k \leq N$, be an integer. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$
\begin{aligned}
&\Box((\Box FrozenBelow(\mathsf{ORD_j}[k], P_m) \wedge FlipCount(\mathsf{ORD_j}[k]) = f) \\
&\quad \rightarrow \Diamond(\quad loc_j = \ell_{8.8} \wedge FlipCount(\mathsf{ORD_j}[k]) \leq f + 1 \\
&\qquad\qquad \wedge (\Box(loc_j \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow (k > |\gamma(a_j)| \vee m \in \rho(choice(ord_j, \gamma(a_j)), k))) \\
&\qquad\qquad\quad \vee\, dist(P_m, P_i) < dist(P_m, P_j) \\
&\qquad\qquad\qquad \rightarrow (\quad \Box(\quad a_j[i] \wedge \mu(ord_j[i], k+1) = \mu(\mathbf{ORD}_{\mathsf{i}}, k+1) \\
&\qquad\qquad\qquad\qquad\qquad \wedge (locj \notin [\ell_{8.5}, \ell_{8.7}] \rightarrow i \notin \rho(choice(ord_j, \gamma(a_j)), k))) \\
&\qquad\qquad\qquad\quad \vee \Box(\quad loc_j \in [\ell_{8.5}, \ell_{8.7}] \vee \neg a_j[i] \\
&\qquad\qquad\qquad\qquad\qquad \vee \mu(ord_j[i], k+1) = \mu(change(ord_j, \gamma(a_j), i), k+1) \\
&\qquad\qquad\qquad\qquad\qquad \vee i \in \rho(choice(ord_j, \gamma(a_j)), k))))) \ .
\end{aligned}
$$

Proof. The proof is similar to the proof of Lemma 7.3.26, and it follows from Lemmata 7.3.17, 7.3.18, 7.3.19, 7.3.20, 7.3.25, and 7.3.26. ■

As a consequence of Lemmata 7.3.19, 7.3.26, and 7.3.27, if registers below register $\mathsf{ORD_i}[k]$ remain frozen with respect to process P_m, then within two flips on register $\mathsf{ORD_i}[k]$ the value recorded in $\mathsf{ORD_i}[k]$ remains unchanged and thus register $\mathsf{ORD_i}[k]$ becomes frozen.

Corollary 7.3.1. *Let P_i and P_m be (possibly crashed) processes and k, $1 \leq k \leq N$, be an integer. If columns of bit-matrix* ORD *below column k are always frozen and process P_m is always active in column k, then the number of flips that occur on register* $\mathsf{ORD_i}[k]$ *in the states in which registers below* $\mathsf{ORD_i}[k]$ *are frozen is at most $2^{dist(P_m, P_i)}$.*

7.3.6 Highest Priority Non-crashed Processes

Consider some program $\mathcal{S}$ consisting of N concurrent processes $P_1, \ldots, P_N$. Below, $\mathcal{S}^i$ denotes the program obtained from $\mathcal{S}$ by removing process P_i from program $\mathcal{S}$. Process P_i is said to be enabled in a state if at least one of P_i's transitions is enabled in that state. This is denoted by predicate $\mathit{Enabled}_i$.

Lemma 7.3.28 (Weak-Until Eventuality)**.** *Let $(W, \leq)$ be a well-founded structure. Let 0 denote a minimal element in W. Let $\delta : W \to (\mathit{States} \to \mathit{Boolean})$ be a predicate, p, q, and r be state predicates, and $\varphi(i)$ be a state predicate parameterized by process P_i. Then, for program $\mathcal{S}$ consisting of processes $P_1, \ldots P_N$, the following proof rule is sound:*

$$\begin{array}{rcll}
\Box(r & \to & \exists w \cdot \delta(w)) & (1)\\
\Box(\delta(0) & \to & (p \vee q)) & (2)\\
\Box(\delta(w) & \to & \circ(\exists v \leq w \cdot \delta(v) \vee q)) & (3)\\
\Box((\delta(w) \wedge w > 0) & \to & \Diamond(\exists v < w \cdot \delta(v) \vee \exists i \cdot \varphi(i) \vee q)) & (4)\\
\multicolumn{4}{l}{\textit{For every process } P_i,}\\
\Box((\varphi(i) \wedge \delta(w) \wedge w > 0) & \to & (\neg \mathit{Enabled}_i \wedge \circ(\exists v < w \cdot \delta(v) \vee \varphi(i) \vee q)) & (5)\\
\mathcal{S}^i \vdash \Box((\varphi(i) \wedge \delta(w) \wedge w > 0) & \to & \Diamond(p\, \mathcal{W}\, (\exists v < w \cdot \delta(v) \vee q))) & (6)\\
\hline
\Box(r & \to & \Diamond(p\, \mathcal{W}\, q)) & .
\end{array}$$

Proof. Consider any computation sequence $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$ of program $\mathcal{S}$. Assume r holds in state s_i. If q holds is some state s_j, $j \geq i$, then $\Diamond(p \mathcal{W} q)$ is trivially satisfied. Therefore, assume $\neg q$ holds in every state s_j, $j \geq i$. Because of (1), there exists $w \in W$ such that $\delta(w)$ holds in s_i. Since $(W, \leq)$ is a well-founded structure and because of (3), there exists state s_k, $k \geq i$, such that $\delta(w) = v$ continuously holds afterwards. If $v = 0$, then from that point onwards p holds because of (2). Otherwise, $v > 0$ holds and, hence, there exists some P_i such that $\varphi(i)$ holds in some state s_m, $m \geq k$, because of (4). Because of (3), $\varphi(i)$ holds in every state s_n, $n \geq m$.

Clause (5) and $\varphi(i)$ then ensure that P_i is disabled in every state s_n, $n \geq m$. Thus, computation sequence $s_m \longrightarrow s_{m+1} \longrightarrow s_{m+2} \longrightarrow \ldots$ is a computation sequence of program $\mathcal{S}^i$. Because of clause (6), $\Diamond(p\,\mathcal{W}\,(\exists v < w \cdot \delta(v) \vee q))$ holds for $\mathcal{S}^i$ and hence for $\mathcal{S}$. Consequently, in this sequence $\Diamond\Box p$ holds. Therefore, $\Diamond(p\,\mathcal{W}\,q)$ is satisfied. ∎

Next we formally introduce the notion of the highest priority non-crashed process. Intuitively, the highest priority non-crashed process (locally in process P_i) is the first non-crashed process selected by function $\mathit{choice}(\mathit{ord}_i, \gamma(a_i))$.

Definition 7.3.9. *Let P_i and P_m be non-crashed processes. We define*

$$\begin{aligned} \mathit{HP}_i(m) \equiv{}& a_i[m] \\ & \wedge ((\mathit{loc}_i \notin [\ell_{8.5}, \ell_{8.7}] \wedge j \in \pi(\mathit{choice}(\mathit{ord}_i, \gamma(a_i)), m)) \rightarrow \mathit{crashed}_j)\ . \end{aligned}$$

We say process P_m is the highest priority non-crashed process (locally in P_i) if predicate $\mathit{HP}_i(m)$ holds.

The following lemma claims that every non-crashed process P_m eventually becomes the highest priority non-crashed process (locally in P_i) unless P_m enters its critical section.

Lemma 7.3.29. *Let P_i and P_m be non-crashed processes. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box((\mathit{loc}_m \in [\ell_1, \ell_{18}] \wedge \mathit{Active}(P_m)) \rightarrow \Diamond(\mathit{HP}_i(P_m)\,\mathcal{W}\,\mathit{InCS}_m))\ .$$

Proof. We use the proof rule in Lemma 7.3.28 to prove the property in this lemma. In the proof rule, we use the following substitutions: $r \equiv \neg\mathit{crashed}_i \wedge \neg\mathit{crashed}_m \wedge \mathit{loc}_m \in [\ell_1, \ell_{18}] \wedge \mathit{Active}(P_m)$, $p \equiv \mathit{HP}_i(P_m)$, and $q \equiv \mathit{InCS}_m$. We assume that $\mathcal{S}$ is the SLEX program in Figures 4.1 and 4.2 consisting of $n = N$ processes. The proof rule in Lemma 7.3.28 involves two state predicates: the ranking predicate $\delta(w)$, for $w \in W$, W being a well-founded structure, and the disabling predicate $\varphi(i)$ parameterized by process P_i. Next we describe those two predicates.

i) Ranking predicate δ.

Let $W = \{0, \ldots, 2^N\}$ with the usual $\leq$ relation be the well-founded structure in Lemma 7.3.28. We define predicate $\delta(w)$, for $w \in W$, as follows: If $InCS_m$ holds in a state, then $\delta(0)$ holds in that state. Otherwise, let P_j be any process, $k = N-n+1$ be the integer identifying the first non-frozen column in bit-matrix ORD, and w be the number of flips that have occurred on register $\mathsf{ORD_j}[k]$ in the states when registers below $\mathsf{ORD_j}[k]$ were frozen with respect to process P_m. By Corollary 7.3.1, there are at most $2^{dist(P_m,P_j)}$ flips on every register $\mathsf{ORD_j}[k]$ before the value of $\mathsf{ORD_j}[k]$ remains unchanged. Consequently, there are at most $\sum_{j=1}^{N} 2^{dist(P_m,P_j)} = 2^N - 1$ flips in column k before column k is frozen. Let f equals 2^N less the number of flips that have occurred in column k. Predicate $\delta(w)$ holds only if $w = f$ holds. In addition, predicate δ incorporates the invariant that is necessary to establish that registers below $\mathsf{ORD_m}[k]$ are frozen with respect to process P_m.

ii) Disabling predicate φ.

Predicate $\varphi(i)$ holds, for process P_i, if P_i is selected in column $k = N - n + 1$ of bit-matrix ord_i and it is crashed. ■

7.3.7 Highest Priority Non-Crashed Process Enters Critical Section

In this section, we show that the highest priority non-crashed process eventually enters its critical section, provided less than ℓ processes are crashed in the critical section. We first show that if columns up to column k are frozen, then eventually all non-crashed processes always select the same processes in the frozen columns.

Lemma 7.3.30. *Let P_i and P_j be non-crashed processes, P_m be a possibly crashed process, and k, $1 \leq k \leq N + 1$, be an integer. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned}&\Box((\Box \mathit{Active}(m) \wedge \Box \mathit{FrozenColsBelow}(k))\\&\quad \rightarrow \Diamond\Box((loc_i \notin [\ell_{8.5}, \ell_{8.7}] \wedge loc_j \notin [\ell_{8.5}, \ell_{8.7}])\\&\qquad\quad \rightarrow (k \le |\gamma(a_i)| + 1 \wedge k \le |\gamma(a_j)| + 1\\&\qquad\qquad \wedge\, \mu(\mathit{choice}(\mathit{ord}_i, \gamma(a_i)), k) = \mu(\mathit{choice}(\mathit{ord}_j, \gamma(a_j)), k)))) \ .\end{aligned}$$

Proof. The property directly follows from Lemmata 7.3.25 and 7.3.26. ■

As a consequence of the property in the above lemma, if a process is recognized by some non-crashed process as always the highest priority non-crashed process, then that process is eventually recognized by all non-crashed processes as always the highest priority non-crashed process.

Corollary 7.3.2. *Let P_i, P_j, and P_m be non-crashed processes. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$\Box(\Box HP_i(m) \rightarrow \Diamond\Box HP_j(m)) \ . \qquad ■$$

Next we show that if some register $\mathsf{VEC_m}$ remains unchanged, then eventually every non-crashed process P_i determines (locally) the same values of set $E_i(m)$.

Lemma 7.3.31. *Let P_i and P_j be non-crashed processes and P_m be a (possibly crashed) process. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box(w = \mathit{LastWrt}(\mathsf{VEC_m}) \bullet \Diamond(E_i(m) = E_j(m)\ \mathcal{W}\ w \neq \mathit{LastWrt}(\mathsf{VEC_m}))) \ .$$

Proof. The property follows from Lemma 7.3.16. ■

If P_m is the highest priority non-crashed process, then every process P_j with a higher priority than P_m is crashed. If P_m never enters its critical section, then eventually every process with a higher priority than P_m is recorded in program variable p_i, for every non-crashed process P_i. Since $p_i = \pi(\mathit{choice}(\mathit{ord}_i, \gamma(a_i)), i)$, it follows that $\pi(\mathit{choice}(\mathit{ord}_i, \gamma(a_i)), m) \subseteq p_i$ holds. If process P_i is crashed, it does not perform any write operation on register $\mathsf{VEC_j}$. Therefore, program variable A_i eventually records the processes indices described in the lemma below.

Lemma 7.3.32. *Let P_m and P_i be non-crashed processes. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned}\Box((\Box HP_i(m) \wedge \Box\neg \mathit{InCS}_m \wedge \mathit{loc}_i \notin [\ell_{8.5}, \ell_{8.7}] \wedge p_i = \pi(\mathit{choice}(\mathit{ord}_i, \gamma(a_i)), m))\\ \to \Diamond\Box(\mathit{loc}_i \notin [\ell_{8.5}, \ell_{8.7}] \to A_i \supseteq \{j \mid \mathit{CrashedInCS}_j \wedge j \in p_i\}\\ \cup \bigcup_{i' \in p_i} E_i(i') \setminus \{i\}))\ . \qquad \blacksquare\end{aligned}$$

From the above lemma and Lemma 7.3.12(b), it immediately follows that the value of variable A_m eventually remains unchanged.

Corollary 7.3.3. *Let P_m be a non-crashed process. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned}\Box((\Box HP_m(m) \wedge \Box\neg \mathit{InCS}_m)\\ \to \Diamond\Box(\mathit{loc}_m \notin [\ell_{8.5}, \ell_{8.7}] \to A_m = \{j \mid \mathit{CrashedInCS}_j \wedge j \in p_m\}\\ \cup \bigcup_{i' \in p_m} E_m(i') \setminus \{m\}))\ . \qquad \blacksquare\end{aligned}$$

Since every non-crashed process P_i eventually identifies process P_m as the highest priority non-crashed process, by Lemma 7.3.32 and Corollary 7.3.3, variable A_i records the values of A_m, provided P_m is distinct from P_i.

Lemma 7.3.33. *Let P_m and P_i be distinct non-crashed processes. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$\begin{aligned}\Box((\Box HP_i(m) \wedge \Box\neg \mathit{InCS}_m)\\ \to \Diamond\Box((\mathit{loc}_i \notin [\ell_{8.5}, \ell_{8.7}] \wedge \mathit{loc}_m \notin [\ell_{8.5}, \ell_{8.7}]) \to A_m \subseteq A_i))\ . \qquad \blacksquare\end{aligned}$$

Because of the above lemma, if at least ℓ processes are recorded in set A_m of the always highest priority non-crashed process P_m, and P_m never enters its critical section, then eventually every distinct non-crashed process P_i sets its register $\mathsf{TRY_i}$ to *false*.

Lemma 7.3.34. *Let P_m and P_i be distinct non-crashed processes. For all computation sequences of program $\mathcal{S}$, the following property holds:*

$$\begin{aligned}&\Box((\Box HP_i(m) \wedge \Box\neg InCS_m \wedge loc_m \notin [\ell_{8.5}, \ell_{8.7}] \wedge |A_m| \geq \ell) \\ &\quad \rightarrow \Diamond(\mathbf{TRY}_{\mathsf{i}} = \mathit{false}\, \mathcal{W}\, |A_m| < \ell)) \ . \qquad \blacksquare\end{aligned}$$

Assume there are less than ℓ processes crashed in the critical section and P_m remains the highest priority non-crashed process. If $|A_m| \geq \ell$, because of Lemmata 7.3.33 and 7.3.34, eventually all non-crashed processes distinct from P_m set their registers $\mathsf{TRY_i}$ to *false*. Consequently, only less than ℓ processes crashed in the critical section record *true* in their registers $\mathsf{TRY_j}$. Hence, eventually $|A_m| < \ell$ holds unless P_m enters its critical section.

Lemma 7.3.35. *Let P_m be a non-crashed process. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$\Box((|\{j \mid \mathit{CrashedInCS}_j\}| < \ell \wedge \Box HP_m(m)) \rightarrow \Diamond(|A_m| < \ell\, \mathcal{W}\, InCS_m)) \ . \qquad \blacksquare$$

Because of the above lemma, register $\mathsf{TRY_m}$ eventually records *true* unless process P_m enters its critical section.

Corollary 7.3.4. *Let P_m be a non-crashed process. For every computation sequence of program $\mathcal{S}$, the following property eventually holds:*

$$\Box((|\{j \mid \mathit{CrashedInCS}_j\}| < \ell \wedge \Box HP_m(m)) \rightarrow \Diamond(\mathsf{TRY_m} = \mathit{true}\, \mathcal{W}\, InCS_m)) \ . \qquad \blacksquare$$

Assume there are less than ℓ processes crashed in the critical section and P_m remains the highest priority non-crashed process. Then by the above corollary, eventually $\mathsf{TRY_m}$ remains *true*. By Lemma 7.3.12(a), if P_m is at location ℓ_{18}, $B_m = E_m(m)$ holds. Because the value of $\mathsf{TRY_m}$ remains unchanged, the value of $\mathsf{VEC_m}$ remains unchanged as well. Hence, by Lemma 7.3.16, the value of $E_m(m)$ remains unchanged. By Lemma 7.3.31, for every non-crashed process P_i, eventually $E_i(m) = E_m(m)$ holds. Since P_m is the highest priority non-crashed process, because of Lemma 7.3.32, eventually $A_i \supseteq E_i(m)$ holds for every non-crashed process P_i distinct from P_m.

Assume that $|B_m| > \ell$ remains to hold. Then eventually $|A_i| > \ell$ and $\mathsf{TRY_i} = \mathit{false}$ always hold for every non-crashed process P_i distinct from P_m. Because there are less than ℓ processes crashed in the critical section, i.e., with $\mathsf{TRY_j} = \mathit{true}$, eventually there are at most ℓ processes in $E_m(m)$, which is a contradiction with the assumption that $|B_m| > \ell$.

Lemma 7.3.36. *Let P_m be a non-crashed process. For all computation sequences of program $\mathcal{S}$, the following property eventually holds:*

$$\Box((|\{j \mid \mathit{CrashedInCS}_j\}| < \ell \wedge \Box \mathit{HP}_m(m)) \rightarrow \Diamond(|B_m| \leq \ell \,\mathcal{W}\, \mathit{InCS}_m)) \ . \qquad \blacksquare$$

The next theorem is a direct consequence of Lemmata 7.3.29 and 7.3.36. Every non-crashed process eventually enters its critical section, provided that less than ℓ processes are crashed in the critical section.

Theorem 7.2. *Let P_i be a process. For all computation sequences of program $\mathcal{S}$, the following property holds:*

$$(|\{j \mid \mathit{CrashedInCS}_j\}| < \ell \wedge \neg \mathit{crashed}_i) \rightarrow \Box\Diamond \mathit{InCS}_i \ . \qquad \blacksquare$$

CHAPTER 8

CONCLUSION

We have developed an assertional correctness proof of a complicated SLEX algorithm. That algorithm is a slight improvement of the SLEX algorithm in (Abraham, Dolev, Herman, and Koll 2001). Both these algorithms utilize single-writer multiple-reader regular registers. They are also able to cope with a limited number of crashed processes. Both the algorithms satisfy the safety property that if at most ℓ processes are crashed in the critical section, from some point onwards at most ℓ processes are simultaneously be in their critical sections. The SLEX algorithm described in the current paper satisfies a stronger liveness property than the SLEX in (Abraham, Dolev, Herman, and Koll 2001). That liveness property is that every non-crashed process eventually enters its critical section, provided that less than ℓ processes are crashed in the critical section. The SLEX algorithm in (Abraham, Dolev, Herman, and Koll 2001) satisfies the liveness property that every non-crashed process eventually enters its critical section only if less than ℓ processes are crashed outside the remainder section.

Our constructive proof has been carried out in Linear–Time Temporal Logic and utilized a history to model access to regular registers. Our analysis has provided some new insight in the correctness of the algorithm. That insight has allowed us to identify some possible improvements of the SLEX algorithm in (Abraham, Dolev, Herman, and Koll 2001). We have also explicitly formulated auxiliary quantities required to establish the program's properties. We have characterized processes (and their minimum number) identified by some process as attempting to enter their critical sections. We have also developed a proof rule for deriving eventuality properties of programs in the presence of disabled processes. We have used that proof rule to structure our correctness proof of the liveness property.

In the future, we will continue in the mechanical verification of our proof. This mechanization has been started using the PVS theorem prover (Owre, Shankar, Rushby, and Stringer-Calvert 2001a; Owre, Shankar, Rushby, and Stringer-Calvert 2001b; Owre, Shankar, Rushby, and Stringer-Calvert 2001c).

APPENDIX A

UNITY PROGRAM

In this appendix, we give a UNITY representation of the SLEX program executed by process P_i, for $i = 1, 2, \ldots, N$.

```
$loc = \ell_1$                  $\longrightarrow$  $loc := \ell_2$
$loc = \ell_2$                  $\longrightarrow$  $\mathsf{invoke\_write}(\mathsf{X_i}, \mathit{true}); loc := \ell_{2.1}$
$loc = \ell_{2.1}$              $\longrightarrow$  $\mathsf{respond\_write}(\mathsf{X_i}); loc := \ell_3$
$loc = \ell_3$                  $\longrightarrow$  $j := 1; loc := \ell_4$
$loc = \ell_4$                  $\longrightarrow$  $\mathsf{invoke\_read}(\mathsf{VEC_j}); loc := \ell_{4.1}$
$loc = \ell_{4.1}$              $\longrightarrow$  $\mathsf{respond\_read}(v[j], \mathsf{VEC_j}); loc := \ell_5$
$loc = \ell_5 \wedge j < N$     $\longrightarrow$  $j := j + 1; loc := \ell_4$
$loc = \ell_5 \wedge j = N$     $\longrightarrow$  $loc := \ell_6$
$loc = \ell_6$                  $\longrightarrow$  $vec := report(v, i); loc := \ell_7$
$loc = \ell_7$                  $\longrightarrow$  $\mathsf{invoke\_read}(\mathsf{TRY_i}); loc := \ell_{7.1}$
$loc = \ell_{7.1}$              $\longrightarrow$  $\mathsf{respond\_read}(old\_try, \mathsf{TRY_i}); loc := \ell_8$
$loc = \ell_8$                  $\longrightarrow$  $loc := \ell_{8.1}$
$loc = \ell_{8.1}$              $\longrightarrow$  $j := 1; loc := \ell_{8.2}$
$loc = \ell_{8.2}$              $\longrightarrow$  $\mathsf{invoke\_read}(\mathsf{TRY_j}); loc := \ell_{8.2.1}$
$loc = \ell_{8.2.1}$            $\longrightarrow$  $\mathsf{respond\_read}(t[j], \mathsf{TRY_j}); loc := \ell_{8.3}$
$loc = \ell_{8.3}$              $\longrightarrow$  $\mathsf{invoke\_read}(\mathsf{X_j}); loc := \ell_{8.3.1}$
$loc = \ell_{8.3.1}$            $\longrightarrow$  $\mathsf{respond\_read}(x[j], \mathsf{X_j}); loc := \ell_{8.4}$
```

$loc = \ell_{8.4}$	$\longrightarrow$	$a[j] := x[j] \vee t[j]; loc := \ell_{8.5}$
$loc = \ell_{8.5} \wedge a[j]$	$\longrightarrow$	$loc := \ell_{8.6}$
$loc = \ell_{8.5} \wedge \neg a[j]$	$\longrightarrow$	$loc := \ell_{8.7}$
$loc = \ell_{8.6}$	$\longrightarrow$	$\mathsf{invoke\_read}(\mathsf{ORD_j}); loc := \ell_{8.6.1}$
$loc = \ell_{8.6.1}$	$\longrightarrow$	$\mathsf{respond\_read}(ord, \mathsf{ORD_j}) : loc := \ell_{8.7}$
$loc = \ell_{8.7} \wedge j < N$	$\longrightarrow$	$j := j + 1; loc := \ell_{8.2}$
$loc = \ell_{8.7} \wedge j = N$	$\longrightarrow$	$loc := \ell_{8.8}$
$loc = \ell_{8.8}$	$\longrightarrow$	$p := \pi(choice(ord, \gamma(a)), i); loc := \ell_{8.9}$
$loc = \ell_{8.9}$	$\longrightarrow$	$A := \{j \mid j \neq i \wedge t[j] \wedge (j \in p \vee$
		$\exists i' \cdot (i' \in p \wedge \neg dominates(v, j, i')))\};$
		$loc := \ell_{8.10}$
$loc = \ell_{8.10}$	$\longrightarrow$	$try := \lvert A \rvert < \ell; loc := \ell_9$
$loc = \ell_9$	$\longrightarrow$	$\mathsf{invoke\_write}(\mathsf{TRY_i}, try); loc := \ell_{9.1}$
$loc = \ell_{9.1}$	$\longrightarrow$	$\mathsf{respond\_write}(\mathsf{TRY_i}); loc := \ell_{10}$
$loc = \ell_{10} \wedge try \wedge \neg old\_try$	$\longrightarrow$	$loc := \ell_{11}$
$loc = \ell_{10} \wedge (\neg try \vee old\_try)$	$\longrightarrow$	$loc := \ell_{12}$
$loc = \ell_{11}$	$\longrightarrow$	$\mathsf{invoke\_write}(\mathsf{VEC_i}, vec); loc := \ell_{11.1}$
$loc = \ell_{11.1}$	$\longrightarrow$	$\mathsf{respond\_write}(\mathsf{VEC_i}); loc := \ell_{12}$
$loc = \ell_{12} \wedge \neg try$	$\longrightarrow$	$loc := \ell_1$
$loc = \ell_{12} \wedge try$	$\longrightarrow$	$loc := \ell_{13}$
$loc = \ell_{13}$	$\longrightarrow$	$j := 1; loc := \ell_{14}$
$loc = \ell_{14}$	$\longrightarrow$	$\mathsf{invoke\_read}(\mathsf{VEC_j}); loc := \ell_{14.1}$
$loc = \ell_{14.1}$	$\longrightarrow$	$\mathsf{respond\_read}(v[j], \mathsf{VEC_j}); loc := \ell_{15}$
$loc = \ell_{15}$	$\longrightarrow$	$\mathsf{invoke\_read}(\mathsf{TRY_j}); loc := \ell_{15.1}$
$loc = \ell_{15.1}$	$\longrightarrow$	$\mathsf{respond\_read}(t[j], \mathsf{TRY_j}); loc := \ell_{16}$
$loc = \ell_{16} \wedge j < N$	$\longrightarrow$	$j := j + 1; loc := \ell_{14}$
$loc = \ell_{16} \wedge j = N$	$\longrightarrow$	$loc := \ell_{17}$

$$
\begin{array}{lcl}
loc = \ell_{17} & \longrightarrow & B := \{j \mid t[j] \wedge \neg dominates(v, j, i)\}; \\
 & & loc := \ell_{18} \\
loc = \ell_{18} \wedge |B| > \ell & \longrightarrow & loc := \ell_1 \\
loc = \ell_{18} \wedge |B| \leq \ell & \longrightarrow & loc := \ell_{19} \\
loc = \ell_{19} & \longrightarrow & \text{Critical Section}; loc := \ell_{20} \\
loc = \ell_{20} & \longrightarrow & \mathsf{invoke\_write}(\mathsf{TRY_i}, false); loc := \ell_{\mathit{20.1}} \\
loc = \ell_{20.1} & \longrightarrow & \mathsf{respond\_write}(\mathsf{TRY_i}); loc := \ell_{21} \\
loc = \ell_{21} & \longrightarrow & \mathsf{invoke\_write}(\mathsf{X_i}, false); loc := \ell_{\mathit{21.1}} \\
loc = \ell_{21.1} & \longrightarrow & \mathsf{respond\_write}(\mathsf{X_i}); loc := \ell_{22} \\
loc = \ell_{22} & \longrightarrow & row := change(ord, \gamma(a), i); loc := \ell_{\mathit{23}} \\
loc = \ell_{23} & \longrightarrow & \mathsf{invoke\_write}(\mathsf{ORD_i}, row); loc := \ell_{23.1} \\
loc = \ell_{23.1} & \longrightarrow & \mathsf{respond\_write}(\mathsf{ORD_i}); loc := \ell_{24} \\
loc = \ell_{24} & \longrightarrow & \text{Remainder Section}; loc := \ell_1
\end{array}
$$

BIBLIOGRAPHY

Abraham, U. (2003). Self-stabilizing timestamps. *Theoretical Computer Science 308*(1–3), 449–515.

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.

Arora, A. and M. Nesterenko (2004). Unifying stabilization and termination in message passing systems. *Distributed Computing*.

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

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

Dijkstra, E. W. (1968). *Cooperating Sequential Processes*, pp. 43–112. Academic Press.

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., 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.

Francez, N. (1986). *Fairness.* Springer-Verlag.

Hadid, R. (2002). Space and time efficient self–stabilizing ℓ-exclusion in tree networks. *Journal of Parallel Distributed Computing 62*(5), 843–864.

Henzinger, T. A. (1990). Half-order modal logic: How to prove real-time properties. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pp. 281–296. ACM Press.

Herlihy, M. P. and J. M. Wing (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*(3), 463–492.

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

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. 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. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers.

Manna, Z. and A. Pnueli (1983). Verification of concurrent programs: A temporal proof system. In J. W. de Bakker and J. van Leeuwen (Eds.), *Foundations of Computer Science IV, Distributed Systems: Part 2*, Volume 159 of *Mathematical Centre Tracts*, pp. 163–255. Center for Mathematics and Computer Science (CWI).

Manna, Z. and A. Pnueli (1991). Completing the temporal picture. *Theoretical Computer Science 83*(1), 97–130.

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.

Owre, S., N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert (2001a). *PVS Language Reference.* Menlo Park, CA: Computer Science Laboratory, SRI International.

Owre, S., N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert (2001b). *PVS Prover Guide.* Menlo Park, CA: Computer Science Laboratory, SRI International.

Owre, S., N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert (2001c). *PVS System Guide.* Menlo Park, CA: Computer Science Laboratory, SRI International.

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

Schmidt, D. A. (1986). *Denotational Semantics — A Methodology for Language Development.* Allyn and Bacon.

Stoy, J. E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press.

ABSTRACT

SELF-STABILIZING ℓ-EXCLUSION:
A CORRECTNESS PROOF

by

MILOSLAV BESTA

August 2005

Advisor: Dr. Frank Stomp

Major: Computer Science

Degree: Doctor of Philosophy

A formal correctness proof of a self-stabilizing ℓ-exclusion algorithm (SLEX) is presented. The analyzed algorithm is an improvement of the SLEX due to Abraham, Dolev, Herman, and Koll, since our version satisfies a stronger liveness property. The proof is formulated in Linear–Time Temporal Logic and utilizes a history to model access to regular registers. The proof consists of a safety part and a liveness part. Our analysis provides some new insight in the correctness of the algorithm:

1. Our proof is constructive. That is, we explicitly formulate auxiliary quantities required to establish some of the properties. This contrasts with the operational arguments of Abraham et al., where many quantities are not explicitly formulated and the validity of the above mentioned properties is established by disproving their non-existence.

2. We characterize processes (and their minimum number) identified by some process as attempting to enter their critical sections.

3. A novel proof rule for reasoning about programs in the presence of disabled processes is presented to structure the liveness proof.

AUTOBIOGRAPHICAL STATEMENT

Miloslav Besta is a Ph.D. candidate in the Department of Computer Science at Wayne State University in Detroit, Michigan. His research interests include several areas of formal methods. In particular, he is interested in formal verification of complex parallel and distributed algorithms and in the use of software tools supporting semi- and fully-automated verification of software. He has received an M.A. degree in Computer Science from Wayne State University, and M.Sc. and B.Sc. degrees in Computer Science from Palacky University in Olomouc, Czech Republic.

References

[1] Besta, M. and F. Stomp: A Complete Mechanization of Correctness of a String-Preprocessing Algorithm. *Formal Methods in System Design 27*(1), pp. 5–27, Kluwer Academic Publishers, 2005.

[2] Besta, M. and F. Stomp: Mechanization of a Proof of String-Preprocessing in Boyer-Moore's Pattern Matching Algorithm. In *Proceedings of the* 8th *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '02)*, pp. 68–77, Greenbelt, MD, December 2002.

[3] Plasil, F., S. Visnovsky, and M. Besta: Bounding Component Behavior via Protocols. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 30)*, pp. 387–398, Santa Barbara, CA, August 1999.

[4] Besta, M.: Description of Component Behavior. In *Proceedings of Week of Doctoral Studies (WDS '99)*, pp. 597–604, Prague, CZ, June 1999.

[5] Sklenar V. and M. Besta: Distributed Objects in Windows. In *Proceedings of Objects '97*, pp. 71–80, Prague, CZ, November 1997.