Correctness Proofs of the Peterson-Fischer Mutual Exclusion Algorithms

by

Christopher P. Colby

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Computer Science and Engineering at the Massachusetts Institute of Technology

June 1989

© Christopher P. Colby, 1989

The author hereby grants to MIT permission to reproduce and to distribute copies of this thesis document in whole or in part.
Correctness Proofs of the Peterson-Fischer Mutual Exclusion Algorithms

by

Christopher P. Colby

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 1989

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

Abstract

The Peterson-Fischer 2-process mutual exclusion algorithm [PF] is introduced in a slightly modified form. An invariant-assertional proof of mutual exclusion is presented for the 2-process algorithm. Next, the Peterson-Fischer n-process mutual exclusion algorithm is introduced conceptually as a tournament of \(\lfloor \lg n \rfloor\) 2-process competitions. A mutual-exclusion proof of the n-process algorithm is presented, based on a mapping between states of the n-process system and states of the 2-process system. This mapping delineates the correspondence between the 2-process code and one iteration (competition) of the n-process code. In this way, the statement of correctness of the 2-process algorithm is used as a lemma for the n-process proof.

Thesis Supervisor: Nancy A. Lynch
Title: Professor, Department of Electrical Engineering and Computer Science
Contents

1 Introduction .............................................. 5

2 The 2-Process Mutual Exclusion Algorithm ................. 6
   2.1 The algorithm ........................................... 6
   2.2 Atomic actions .......................................... 6
   2.3 The 2-process algorithm satisfies mutual exclusion .... 9

3 The n-Process Mutual Exclusion Algorithm .................. 15
   3.1 The algorithm ........................................... 15
   3.2 Atomic actions .......................................... 15
   3.3 The n-process algorithm satisfies mutual exclusion .... 21

4 Conclusion .................................................. 33
List of Figures

2-1 The Peterson-Fischer 2-process mutual exclusion algorithm. ..................... 7
2-2 Behaviors of the atomic actions of the Peterson-Fischer 2-process algorithm. .. 8
2-3 The Atomic Steps of the 2-process algorithm. ........................................ 8
2-4 The Peterson-Fischer 2-process mutual exclusion algorithm using only the Atomic Steps. The flow of control is defined by the value of $PC[i]$. ......................... 9

3-1 The Peterson-Fischer $n$-process mutual exclusion algorithm. ...................... 16
3-2 Behaviors of the atomic actions of the Peterson-Fischer $n$-process algorithm. .. 18
3-3 The Atomic Steps of the $n$-process algorithm. ...................................... 19
3-4 The Peterson-Fischer $n$-process mutual exclusion algorithm shown as atomic steps. The flow of control is defined by the value of $PC[i]$. ......................... 20
3-5 Possible cases for $a_t$. ................................................................. 27
Chapter 1

Introduction

This paper presents a proof that the Peterson-Fischer mutual exclusion algorithm [PF], shown in Figure 3-1, satisfies mutual exclusion. Intuitively, the algorithm operates as a single-elimination tournament between the n processes, where each process must win \( \lceil \lg n \rceil \) competitions with other processes. The Peterson-Fischer 2-process mutual exclusion algorithm, shown in Figure 2-1, outlines a single such competition and is the building block for the n-process algorithm.

The approach that we will take to prove the correctness of the n-process algorithm in Chapter 3 is to simplify it to the point at which we can map it to the 2-process algorithm. At that point, a correctness proof of the 2-process algorithm will suffice to complete the proof.

Since the 2-process algorithm has a finite number of possible states, a straightforward way to prove that it satisfies mutual exclusion is to mechanically enumerate all of its reachable states. Then, they all may be examined to conclude that in no reachable state are both processes in the Critical region. This is the approach taken by Peterson and Fischer in [PF]. In this paper, we take in Chapter 2 an alternative approach—an invariant-assertional proof.
Chapter 2

The 2-Process Mutual Exclusion Algorithm

2.1 The algorithm

The Peterson-Fischer 2-process mutual exclusion algorithm is based on the idea of a competition between two processes, where \( p_0 \) and \( p_1 \) are opponents. The algorithm is shown in Figure 2-1.

2.2 Atomic actions

Let us define the atomic actions of the 2-process algorithm to be any invocation of (i.e., a READ from or a WRITE to) a shared variable (i.e., \( q[0] \) or \( q[1] \)). Now, we can define two non-shared variables (i.e., local to the processes), \( t \) and \( PC \), both indexed by \( \{0, 1\} \), to completely define the behavior of each individual atomic action. These behaviors are shown in Figure 2-2. Using those, we can rewrite the algorithm such that each step of the algorithm is atomic. (We will call these Atomic Steps.) This version of the algorithm is shown in Figure 2-4. As described in Figure 2-2, the flow of control is defined by \( PC[i] \)—if \( PC[i] = x \), then Atomic Step \( x \) is the next step that \( p_i \) will execute. The actual Atomic Steps are shown in Figure 2-3.

The algorithm shown in Figure 2-4 is actually a slight extension over the algorithm shown in Figure 2-1. The algorithm as written in Figure 2-1 would require that the initial value of \( PC[0] \) is 0 and the initial value of \( q[0] \) is \( \text{nil} \). The algorithm shown in Figure 2-4 imposes no constraints
Shared variables:

- \( q \): an array indexed by \{0, 1\} of values from \{\text{nil}, \text{T} = 1, \text{F} = 0\}, initially \langle \text{nil}, \text{nil} \rangle, where \(q[i]\) is written by \(p_i\) and read by both

Notation: \(\text{opp}(i) = -i\)  **the opponent of \(i\)**

Code for \(p_i\):

\[
\begin{align*}
q[i] &\leftarrow \text{if } q[\text{opp}(i)] = \text{nil} \text{ then } \text{T else } i \oplus q[\text{opp}(i)] \\
q[i] &\leftarrow \text{if } q[\text{opp}(i)] = \text{nil} \text{ then } q[i] \text{ else } i \oplus q[\text{opp}(i)] \\
\text{wait until } q[\text{opp}(i)] = \text{nil or } (i \oplus (q[\text{opp}(i)] \neq q[i]))
\end{align*}
\]

**Critical region**

\(q[i] \leftarrow \text{nil}\)

**Remainder region**

Figure 2-1: The Peterson-Fischer 2-process mutual exclusion algorithm.

on the initial value of \(PC[0]\) (and \(t[0]\)) and requires only that \(q[0]\) be \text{nil} iff \(PC[0] \leq 1\). This is the form of the algorithm that we will prove correct. Obviously, if we show the algorithm in Figure 2-4 to be correct, we have shown the algorithm in Figure 2-1 to be correct. This is true because the set of initial states of the algorithm in Figure 2-1 is a subset of the set of allowable initial states of the algorithm in Figure 2-4.

An execution \(\alpha\) of the system is a sequence \(s_0a_0s_1a_1\ldots\), either finite or infinite. Each \(a_t\) is an atomic action taken by either \(p_0\) or \(p_1\). Each \(s_t\) is a state of the 2-process system—an ordered triple \((PC, q, t)\), where \(PC\), \(q\), and \(t\) are arrays indexed by \{0, 1\}. A schedule \(\beta\) of the execution \(\alpha\) is the sequence \(s_0s_1\ldots\). Note that \(\alpha\) is uniquely defined by \(\beta\). In the following mutual exclusion correctness proof of the Peterson-Fischer 2-process algorithm, we will consider all possible schedules \(\beta\).

The following conventions will be used when discussing states of the system: For a state \(s_t = (\langle PC_0, PC_1 \rangle, \langle q_0, q_1 \rangle, \langle t_0, t_1 \rangle)\), \(s_t.PC[i] = PC_i\), \(s_t.q[i] = q_i\), and \(s_t.t[i] = t_i\). Also, \(s_{t_1} = s_{t_2}\) iff all six elements of \(s_{t_1}\) are equal to the corresponding six elements of \(s_{t_2}\). If an element’s value is said to be \(\ast\), then it’s value does not matter (i.e., it may take on any value without affecting
READ by $p_i$ (of $q[\text{opp}(i)]$):

- If $q[\text{opp}(i)] \neq \text{nil}$, then $t[i] \leftarrow q[\text{opp}(i)]$
- $PC[i] \leftarrow$ label of next Atomic Step to be executed by $p_i$

WRITE by $p_i$ of value $v$ (into $q[i]$):

- $q[i] \leftarrow v$
- $t[i] \leftarrow \text{nil}$
- $PC[i] \leftarrow$ label of next Atomic Step to be executed by $p_i$

Figure 2-2: Behaviors of the atomic actions of the Peterson-Fischer 2-process algorithm.

Atomic Step 0:
if $q[\text{opp}(i)] \neq \text{nil}$
then $t[i] \leftarrow q[\text{opp}(i)]$
$PC[i] \leftarrow 1$

Atomic Step 1:
if $t[i] = \text{nil}$
then $q[i] \leftarrow T$
else $q[i] \leftarrow i \oplus t[i]$
$t[i] \leftarrow \text{nil}$
$PC[i] \leftarrow 2$

Atomic Step 2:
if $q[\text{opp}(i)] \neq \text{nil}$
then $t[i] \leftarrow q[\text{opp}(i)]$
$PC[i] \leftarrow 3$

Atomic Step 3:
if $t[i] \neq \text{nil}$
then $q[i] \leftarrow i \oplus t[i]$
else $q[i] \leftarrow q[i]$
$t[i] \leftarrow \text{nil}$
$PC[i] \leftarrow 4$

Atomic Step 4:
if $q[\text{opp}(i)] = \text{nil}$ or $(i \oplus (q[\text{opp}(i)] \neq q[i]))$
then $PC[i] \leftarrow 5$
else $PC[i] \leftarrow 4$

Atomic Step 5:
$q[i] \leftarrow \text{nil}$
$t[i] \leftarrow \text{nil}$
$PC[i] \leftarrow 0$

Figure 2-3: The Atomic Steps of the 2-process algorithm.
Shared variables:
- \( q \): an array indexed by \{0, 1\} of values from \{\text{nil}, T = 1, F = 0\}, initially \( q_0 = \text{nil} \) if \( PC_0 \leq 1 \) (see below), and \( q[0] \in \{T, F\} \) otherwise. Variable \( q[i] \) is written by \( p_i \) and read by both.

Local variables:
- \( t \): an array indexed by \{0, 1\} of values from \{\text{nil}, T = 1, F = 0\}, initially \( t_0 \) may be any value. Variable \( t[i] \) is written and read only by \( p_i \).
- \( PC \): an array indexed by \{0, 1\} of values from \{0, 1, \ldots, 5\}, initially \( PC_0 \) may be any value. Variable \( PC[i] \) is written only by \( p_i \) and never read.

Notation: \( \text{opp}(i) = \neg i \) **the opponent of \( i \)**

Code for \( p_i \): At every step, execute Atomic Step \( PC[i] \). The actual Atomic Steps are shown in Figure 2-3.

Definitions:
- \( p_i \) is in the Remainder region iff \( PC[i] = 0 \).
- \( p_i \) is in the Critical region iff \( PC[i] = 5 \).

Figure 2-4: The Peterson-Fischer 2-process mutual exclusion algorithm using only the Atomic Steps. The flow of control is defined by the value of \( PC[i] \).

the truth of the statement).

2.3 The 2-process algorithm satisfies mutual exclusion

In this section, we will show that the Peterson-Fischer 2-process mutual exclusion algorithm indeed does satisfy mutual exclusion. The approach will be to consider any possible schedule \( \beta \) of the system and show that it is not possible for any state \( s \) in \( \beta \) to exhibit \( s.PC[0] = 5 \) and \( s.PC[1] = 5 \).

Lemma 2.1 For all reachable states of the 2-process algorithm, \( PC[i] \) is either 0 or 1 iff \( q[i] = \text{nil} \). I.e.:

\[
\forall t \geq 0 \forall i = 0, 1 [s_t.PC[i] \leq 1 \iff s_t.q[i] = \text{nil}]
\]

Proof: The statement is true for the initial state by definition of the algorithm in Figure 2-4. Also, Atomic Step 5 of \( p_i \) is the the only action that sets \( q[i] \) to \text{nil}, and it is the only action
that sets $PC[i]$ to 0. Furthermore, Atomic Step 0 of $p_i$ does not change $q[i]$, and it is the only action that sets $PC[i]$ to 1. Finally, atomic Step 1 of $p_i$ always sets $q[i]$ to a non-nil value, and it is the only action that sets $PC[i]$ to 2.

Before we continue with state invariants, we will first make some statements about the transition from one reachable state to the next.

**Lemma 2.2** Let $\alpha = a_0 a_1 a_2 \ldots$ be an execution of the Peterson-Fischer 2-process algorithm, and let $t$ be any index such that $s_{t+1}$ occurs in $\alpha$. Then, the following are true:

1. If $s_t.PC[0] = s_{t+1}.PC[0]$ and $s_t.PC[1] = s_{t+1}.PC[1]$, then $s_t = s_{t+1}$.

2. If $s_t.PC[i] \neq s_{t+1}.PC[i]$, then $s_t.PC[\text{opp}(i)] = s_{t+1}.PC[\text{opp}(i)]$.

3. If $s_t.PC[i] = 4$ and $s_{t+1}.PC[i] = 5$ then
   
   (a) $s_t.q[0] = s_{t+1}.q[0]$
   
   (b) $s_t.q[1] = s_{t+1}.q[1]$
   
   (c) Either $s_t.q[\text{opp}(i)] = \text{nil}$ or $i \oplus (s_t.q[\text{opp}(i)] \neq s_t.q[i])$
   
   (d) Either $s_{t+1}.q[\text{opp}(i)] = \text{nil}$ or $i \oplus (s_{t+1}.q[\text{opp}(i)] \neq s_{t+1}.q[i])$

   (e) If $s_t.PC[\text{opp}(i)] \geq 2$, then $i \oplus (s_t.q[\text{opp}(i)] \neq s_t.q[i])$ and $i \oplus (s_{t+1}.q[\text{opp}(i)] \neq s_{t+1}.q[i])$

   (f) If $s_{t+1}.PC[\text{opp}(i)] \geq 2$, then $i \oplus (s_t.q[\text{opp}(i)] \neq s_t.q[i])$ and $i \oplus (s_{t+1}.q[\text{opp}(i)] \neq s_{t+1}.q[i])$

4. If $s_{t+1}.PC[i] \in \{1, 3\}$ and $s_{t+1}.t[i] \neq s_{t+1}.q[\text{opp}(i)]$, then $s_t.PC[i] = s_{t+1}.PC[i]$.

**Proof:**

1. Action $a_t$ of the execution that the schedule $\beta$ defines must, by definition, be either an Atomic Step by $p_0$ or an Atomic Step by $p_1$. The only Atomic Step that can possibly leave both $PC[0]$ and $PC[1]$ unchanged is Atomic Step 5. Now, if Atomic Step 5 left both $PC[0]$ and $PC[1]$ unchanged, then it must have executed the then branch, and thus altered none of the six state elements.
2. $PC[i]$ can be changed only by Atomic Steps of $p_i$. Thus, if an Atomic Step changes $PC[i]$, it cannot change $PC[opp(i)]$.

3. These statements come directly from examination of Atomic Step 4 and from Lemma 2.1.

4. This comes from examination of Atomic Step 1 and Atomic Step 3.

Now we continue with the state invariants.

**Lemma 2.3** For all reachable states of the 2-process algorithm, if $PC[0]$ is either 1 or 3, and $PC[1]$ is either 1 or 3, then either $t[0] = q[1]$ or $t[1] = q[0]$. I.e.:

$$\forall t \geq 0 [\forall j = 0, 1 [s_t.PC[j] \in \{1, 3\}] \implies \exists i = 0, 1 [s_t.t[i] = s_t.q[opp(i)]]]$$

*Proof:* By induction on the length of the execution. It is obviously true for $s_0$, since $s_0.PC[1] = 0$. Assume it is true for $s_{t-1}$. Now, proceed by contradiction—assume that it is not true for $s_t$. In that case, $\forall i = 0, 1 [s_t.PC[i] \in \{1, 3\} \wedge s_t.t[i] \neq s_t.q[opp(i)]]$. By Part 4 of Lemma 2.2, $s_{t-1}.PC[0] = s_t.PC[0]$ and $s_{t-1}.PC[1] = s_t.PC[1]$. But then by Part 1 of Lemma 2.2, $s_t = s_{t-1}$. However, this is impossible, since our inductive hypothesis states that the Lemma is true for $s_{t-1}$, but our assumption of contradiction states that the Lemma is not true for $s_t$. Thus, the proof is established.

**Lemma 2.4** For all reachable states of the 2-process algorithm in which, for some process $p_i$ and for some $p \in \{T, F\}$, $PC[i] = 3$, $q[opp(i)] = p$, and $t[i] = \neg p$, the following must be true:

1. If $PC[opp(i)] = 2$, then $q[i] \neq i \oplus p$.
2. If $PC[opp(i)] = 3$, then $t[opp(i)] \neq i \oplus p$.
3. If $PC[opp(i)] = 4$, then $q[i] \neq i \oplus p$.
4. $PC[opp(i)] \neq 5$.

*Proof:* By induction on the length of the execution $\alpha = s_0a_0s_1a_1 \ldots$. Since $PC[1] = 0$ in the initial state, all four statements are trivially true for $s_0$. Assume that they are also true for $s_{t-1}$. We proceed by contradiction. Assume that for some $i$ and $p$, $s_t.PC[i] = 3$, $s_t.q[opp(i)] = p$, and $s_t.t[i] = \neg p$. Furthermore, assume that one of the following is true:
1. \( s_t.PC[\text{opp}(i)] = 2 \) and \( s_t.q[i] = i \oplus p \).

2. \( s_t.PC[\text{opp}(i)] = 3 \) and \( s_t.t[\text{opp}(i)] = i \oplus p \).

3. \( s_t.PC[\text{opp}(i)] = 4 \) and \( s_t.q[i] = i \oplus p \).

4. \( s_t.PC[\text{opp}(i)] = 5 \).

By our inductive hypothesis, we know that \( s_t \neq s_{t-1} \). So, by Part 1 of Lemma 2.2, either \( s_{t-1}.PC[0] \neq s_t.PC[0] \) or \( s_{t-1}.PC[1] \neq s_t.PC[1] \). Furthermore, since \( s_t.PC[i] = 3 \) and \( s_t.t[i] \neq s_t.q[\text{opp}(i)] \), it follows from Part 4 of Lemma 2.2 that \( s_{t-1}.PC[i] = s_t.PC[i] = 3 \). (And thus, \( s_{t-1}.t[i] = s_t.t[i] = \neg p \) and \( s_{t-1}.q[i] = s_t.q[i] \).) So, \( s_{t-1}.PC[\text{opp}(i)] \neq s_t.PC[\text{opp}(i)] \). Thus, we have the following four cases for action \( a_{t-1} \), corresponding to the above possible assumptions:

1. \( a_{t-1} \) was Atomic Step 1: \( s_{t-1}.PC[\text{opp}(i)] = 1 \) and \( s_{t-1}.q[i] = s_t.q[i] = i \oplus p \).

2. \( a_{t-1} \) was Atomic Step 2: \( s_{t-1}.PC[\text{opp}(i)] = 2 \) and \( s_{t-1}.t[\text{opp}(i)] = i \oplus p \).

3. \( a_{t-1} \) was Atomic Step 3: \( s_{t-1}.PC[\text{opp}(i)] = 3 \) and \( s_{t-1}.q[i] = s_t.q[i] = i \oplus p \).

4. \( a_{t-1} \) was Atomic Step 4: \( s_{t-1}.PC[\text{opp}(i)] = 4 \).

We will now examine each case separately and show how each leads to a contradiction.

1. By Lemma 2.1, \( s_{t-1}.q[\text{opp}(i)] = \text{nil} \). Examination of Atomic Step 1 reveals that \( s_{t-1}.t[\text{opp}(i)] \) is either \( \text{opp}(i) \oplus p = \neg i \oplus p \) or \( \text{nil} \). In both cases, \( s_{t-1}.q[\text{opp}(i)] \neq s_{t-1}.t[i] \) and \( s_{t-1}.q[i] \neq s_{t-1}.t[\text{opp}(i)] \). Since \( s_{t-1}.PC[i] = 3 \) and \( s_{t-1}.PC[\text{opp}(i)] = 1 \), Lemma 2.3 states that \( s_{t-1} \) is unreachable. This is a contradiction.

2. Since \( s_t.t[\text{opp}(i)] = i \oplus p \), examination of Atomic Step 2 reveals that \( s_{t-1}.q[i] = i \oplus p \). Also, examination of Atomic Step 2 shows that \( s_{t-1}.q[\text{opp}(i)] = s_t.q[\text{opp}(i)] = p \). So, in summary, the following are true for state \( s_{t-1} \): \( PC[i] = 3 \), \( PC[\text{opp}(i)] = 2 \), \( q[i] = i \oplus p \), \( q[\text{opp}(i)] = p \), and \( t[i] = \neg p \). However, by our inductive hypothesis, this cannot be true. (State \( s_{t-1} \) violates Statement 1.) Thus, this is a contradiction.

3. Examination of Atomic Step 3 reveals that there are two possible cases for \( s_{t-1}.q[\text{opp}(i)] \) and \( s_{t-1}.t[\text{opp}(i)] \):
(a) \( s_{t-1}.q[opp(i)] = p \) and \( s_{t-1}.t[opp(i)] = nil \). In this case, \( s_{t-1}.q[opp(i)] \neq s_{t-1}.t[i] \) and \( s_{t-1}.q[i] \neq s_{t-1}.t[opp(i)] \). Since \( s_{t-1}.PC[i] = 3 \) and \( s_{t-1}.PC[opp(i)] = 3 \), Lemma 2.3 states that \( s_{t-1} \) is unreachable. This is a contradiction.

(b) \( s_{t-1}.t[opp(i)] = opp(i) + p = -i + p \). Now, let \( i' = opp(i) \) and \( p' = i + p \). Then, the following are true for state \( s_{t-1} \): \( PC[i'] = 3 \), \( PC[opp(i')] = 3 \), \( q[i'] = p' \), \( t[i'] = -p' \), and \( t[opp(i')] = i' + p' \). However, by our inductive hypothesis, this cannot be true. (State \( s_{t-1} \) violates Statement 2.) Thus, this is a contradiction.

4. By Part 3 of Lemma 2.2, \( s_{t-1}.q[opp(i)] = s.t.q[opp(i)] = p \). Also by Part 3 of Lemma 2.2, \( opp(i) = (s_{t-1}.q[i] \neq s_{t-1}.q[opp(i)]) \). So, \( -i \oplus (s_{t-1}.q[i] \neq p) \). Thus, \( s_{t-1}.q[i] = i \oplus p \). So, in summary, the following are true for state \( s_{t-1} \): \( PC[i] = 3 \), \( PC[opp(i)] = 4 \), \( q[i] = i \oplus p \), \( q[opp(i)] = p \), and \( t[i] = -p \). However, by our inductive hypothesis, this cannot be true. (State \( s_{t-1} \) violates Statement 3.) Thus, this is a contradiction.

Thus, we see that in each case, our assumption that such a state \( s_t \) existed was flawed. Therefore, the proof is established.

**Lemma 2.5** For all reachable states of the 2-process algorithm in which, for some process \( p \) and for some \( p \in \{T, F\} \), \( PC[i] = 4 \), \( PC[opp(i)] = 5 \), and \( q[i] = p \), it must be true that \( q[opp(i)] \neq -i \oplus p \).

**Proof:** By induction on the length of the execution \( \alpha = s_0a_0s_1a_1 \ldots \). Since \( PC[1] = 0 \) in the initial state, this is trivially true for \( s_0 \). Assume that it is also true for \( s_{t-1} \). We proceed by contradiction. Assume that for some \( i \) and \( p \), \( s_t.PC[i] = 4 \), \( s_t.PC[opp(i)] = 5 \), \( s_t.q[i] = p \), and \( s_t.q[opp(i)] = -i \oplus p \). By our inductive hypothesis, we know that \( s_t \neq s_{t-1} \). So, by Part 1 of Lemma 2.2, either \( s_{t-1}.PC[0] \neq s_t.PC[0] \) or \( s_{t-1}.PC[1] \neq s_t.PC[1] \). Furthermore, by Part 3 of Lemma 2.2, \( s_{t-1}.PC[opp(i)] \neq 4 \) (i.e., \( s_{t-1}.PC[opp(i)] = s_t.PC[opp(i)] = 5 \)). So, \( s_{t-1}.PC[i] = 3 \), and action \( a_{t-1} \) is Atomic Step 3. Since \( s_{t-1}.PC[opp(i)] = s_t.PC[opp(i)] \), it follows that \( s_{t-1}.q[opp(i)] = s_t.q[opp(i)] = -i \oplus p \). Now, Atomic Step 3 reveals that there are two cases for \( s_{t-1}.q[i] \) and \( s_{t-1}.t[i] \):

1. \( s_{t-1}.q[i] = p \) and \( s_{t-1}.t[i] = nil \). Assume \( s_{t-2} \neq s_{t-1} \). By Part 3 of Lemma 2.2, \( s_{t-2}.PC[opp(i)] \neq 4 \) (i.e., \( s_{t-2}.PC[opp(i)] = 5 \)). Also, by Part 4 of Lemma 2.2, \( s_{t-2}.PC[i] = 3 \). So, by Lemma 2.2, \( s_{t-2} = s_{t-1} \). This is a contradiction.
2. \( s_{t-1}.q[\text{opp}(i)] = -i \oplus p \) and \( s_{t-1}.t[i] = i \oplus p \). Let \( p' = -i \oplus p \). Then, in summary, the following are true for state \( s_{t-1} \): \( PC[i] = 3 \), \( PC[\text{opp}(i)] = 5 \), \( q[\text{opp}(i)] = p' \), and \( t[i] = -p' \).

By Statement 4 of Lemma 2.4, \( s_{t-1} \) is an unreachable state. This is a contradiction.

Thus, for each case, our original assumption that such a state \( s_t \) existed was flawed. Therefore, the proof is established.

\[\]

**Lemma 2.6** For all reachable states of the 2-process algorithm, either \( PC[0] \neq 5 \) or \( PC[1] \neq 5 \). I.e.:

\[
\forall t \geq 0 [s_t.PC[0] \neq 5 \lor s_t.PC[1] \neq 5].
\]

**Proof:** By induction on the length of the execution \( \alpha = s_0a_0s_1a_1\ldots \). Since \( PC[1] = 0 \) in the initial state, this is trivially true for \( s_0 \). Assume that it is also true for \( s_{t-1} \). We proceed by contradiction. Assume that \( s_t.PC[0] = 5 \) and \( s_t.PC[1] = 5 \). By our inductive hypothesis, we know that \( s_t \neq s_{t-1} \). Then, by Part 1 of Lemma 2.2, \( s_{t-1}.PC[i] \neq s_t.PC[i] \), for some \( i \). Fix \( i \) with this property. Then, \( s_{t-1}.PC[i] = 4 \). By Lemma 2.1, \( s_{t-1}.q[i] \neq \text{nil} \). So, let \( p = s_{t-1}.q[i] \). By Part 3 of Lemma 2.2, \( i \oplus (s_{t-1}.q[\text{opp}(i)] \neq s_{t-1}.q[i]) \). So, \( i \oplus (s_{t-1}.q[\text{opp}(i)] \neq p) \). Thus, \( s_{t-1}.q[\text{opp}(i)] = -i \oplus p \). But then, Lemma 2.5 states that \( s_{t-1} \) is unreachable. This is a contradiction. Therefore, the proof is established.

**Theorem 2.7** The Peterson-Fischer 2-process mutual exclusion algorithm satisfies mutual exclusion.

**Proof:** For any schedule \( \beta \) of the algorithm, \( s_0 = ((0, 0), (\text{nil}, \text{nil}), (\text{nil}, \text{nil})) \). By Lemma 2.6, for no state \( s \) in \( \beta \) is \( s.PC[0] = 5 \) and \( s.PC[1] = 5 \). Thus, by definition, there is no reachable state in which both \( p_0 \) and \( p_1 \) are in the Critical region. Therefore, the Peterson-Fischer 2-process mutual exclusion algorithm satisfies mutual exclusion.
Chapter 3

The $n$-Process Mutual Exclusion Algorithm

3.1 The algorithm

The Peterson-Fischer $n$-process mutual exclusion algorithm is built from the 2-process tournament model. Conceptually, each process must go through $[\lg n]$ competition, arranged in a single-elimination configuration, to move from the Remainder region to the Critical region. The algorithm is shown in Figure 3-1. Each iteration of the loop corresponds to one competition.

3.2 Atomic actions

For the $n$-process algorithm, we will define atomic actions, the behavior of the atomic actions on introduced local variables, and a state of the system in much the same way as we did for the 2-process algorithm in Section 2.2.

Let us define the atomic actions of the $n$-process algorithm to be any invocation of (i.e., a READ from or a WRITE to) a shared variable (i.e., some $q[i]$). Now, we can define four non-shared variables (i.e., local to the processes), $t$, $PC$, $k$, and $op$, all indexed by $\{1, \ldots, n\}$, to completely define the behavior of each individual atomic action. These behaviors are shown in Figure 3-2. Using those, we can rewrite the algorithm such that each step of the algorithm is atomic. (We will call these Atomic Steps.) This version of the algorithm is shown in Figure 3-4.
Shared variables:

- $q$: an array indexed by $\{1, \ldots, n\}$ of pairs $(\text{level}, \text{flag})$, where level is an integer and flag takes on values in $\{T, F\}$. Initially, $q[i] = (0, F)$ for all $i$. Variable $q[i]$ is written by $p_i$ and read by all.

Notation:

- The function $\text{bit}(i, k)$ tells what role $p_i$ plays in level $k$ competition; roles obtainable from binary representation. That is, $\text{bit}(i, k) = \text{bit number } \lfloor \log n \rfloor - k + 1$ of the binary representation of $i$.
- Let $\text{opponents}(i, k)$ denote all potential opponents for $p_i$ at level $k$. Let $\text{opponents}(i, 0) = \emptyset$, for all $i$.

Subroutine $\text{OPP}(i, k)$: (Purpose: to search for opponent.)

```plaintext
for $j \in \text{opponents}(i, k)$ do
    opp ← $q[j]$
    if $\text{level}(\text{opp}) \geq k$ then return (opp)
return (0, F)
```

Code for $p_i$

```plaintext
\begin{align*}
\text{for } k = 1, \ldots, \lfloor \log n \rfloor \text{ do} \\
\text{opp ← } \text{OPP}(i, k) \\
q[i] ← \text{if } \text{level}(\text{opp}) = k \text{ then } (k, \text{bit}(i, k) \oplus \text{flag}(\text{opp})) \text{ else } (k, T) \\
\text{opp ← OPP}(i, k) \\
q[i] ← \text{if } \text{level}(\text{opp}) = k \text{ then } (k, \text{bit}(i, k) \oplus \text{flag}(\text{opp})) \text{ else } q[i] \\
\text{L: opp ← OPP}(i, k) \\
\text{if } (\text{level}(\text{opp}) = k \text{ and } (\text{bit}(i, k) \oplus \text{flag}(\text{opp}) = \text{flag}(q[i]))) \text{ or } \text{level}(\text{opp}) > k \text{ then goto L} \\
\text{**Critical region**} \\
q[i] ← (0, F) \\
\text{**Remainder region**}
\end{align*}
```

Figure 3-1: The Peterson-Fischer n-process mutual exclusion algorithm.

16
As described in Figure 3-2, the flow of control is defined by $PC[i]$. If $PC[i] = x$, then Atomic Step $x$ is the next step that $p_i$ will execute. The actual Atomic Steps are shown in Figure 3-3.

The behaviors of the Atomic Steps, shown in Figure 3-2, deserve some additional explanation. The value of $op[i]$ is the set of all indeces whose corresponding $q$ variable is to be READ during a call to the $OPP(i, k)$ subroutine. In other words, when $OPP(i, k)$ is called (by $p_i$), $op[i]$ gets the value of opponents$(i, k)$. It is the set through which $p_i$ will iterate in its following READs (corresponding to the for loop of $OPP(i, k)$). So, every time $p_i$ has to do a READ, a value is picked arbitrarily and removed from $op[i]$. This value is the index of the variable in $q$ that $p_i$ will read. If $op[i]$ becomes $\emptyset$, then the for loop of $OPP(i, k)$ has been exhausted. This case deserves special consideration: Note that $t[i]$ is set only in READs of variables whose level is sufficiently large. This corresponds to the if in $OPP(i, k)$. Thus, the setting of $t[i]$ corresponds to the return inside the for loop of $OPP(i, k)$. However, what if no member of opponents$(i, k)$ has a sufficiently large level? In this case, $OPP(i, k)$ does an explicit return of $(0, F)$. In the Atomic Step version, though, $t[i]$ is never explicitly set to $(0, F)$. Instead, it is guaranteed to be $(0, F)$ before every sequence of READs (i.e., before every call to $OPP(i, k)$). In this manner, if no opponent's level is high enough, $t[i]$ will never change and thus will be $(0, F)$ after the sequence of READs (i.e., after the call to $OPP(i, k)$). This is the reason that the initial state of $t[i]$ is $(0, F)$ and that every WRITE sets $t[i]$ to $(0, F)$.

Atomic Step $(4, j)$ is a bit complicated. The first then means that the level of $q[j]$ was not sufficiently large and opponents$(i, k)$ has not yet been exhausted. So, just like the other READs, it chooses another element of $op[i]$ and does another READ. If the else branch was taken instead, then the analogous call to $OPP(i, k)$ has terminated. In this case, there are two cases:

1. $OPP(i, k)$ returned $(0, F)$. In this case, $p_i$ does not perform the "goto L" and thus has won the competition. Analogously in Atomic Step $(4, j)$, level$(q[j]) < k$ as shown in the first clause of the second if. Subsequently, $p_i$ executes the second then and thus wins the competition (i.e., increments $k[i]$ and either starts another competition at some $(0, m)$ or progresses to the Critical region at $(5, 0)$).

2. $OPP(i, k)$ returned $q[j]$. In this case, $p_i$ performs a test to determine if $p_i$ has won the competition. Analogously in Atomic Step $(4, j)$, the same test is done in the second clause.
READ by $p_i$ of $q[j]$: 
- $op[i] \leftarrow op[i] \setminus \{j\}$
- if level($q[j]$) $\geq k[i]$ then $t[i] \leftarrow q[j]$
- if first($PC[i]$) = 4 and the next value (shown immediately below) of first($PC[i]$) $\in \{0, 5\}$, then $k[i] \leftarrow k[i] + 1$
- $PC[i] \leftarrow$ label of next Atomic Step to be executed by $p_i$
- if the next Atomic Step is a READ, then $op[i] \leftarrow$ opponents($i, k[i]$)

WRITE by $p_i$ of value $v$ (into $q[i]$): 
- $q[i] \leftarrow v$
- $t[i] \leftarrow (0, F)$
- if first($PC[i]$) = 5 then $k[i] = 1$
- $PC[i] \leftarrow$ label of next Atomic Step to be executed by $p_i$
- if the next Atomic Step is a READ, then $op[i] \leftarrow$ opponents($i, k[i]$)

Figure 3-2: Behaviors of the atomic actions of the Peterson-Fischer $n$-process algorithm.

of the second if. If it fails, the last else is taken, and another sequence of READs is started (i.e., OPP($i, k$) is called again).

An execution $\alpha$ of the system is a sequence $S_0a_0S_1a_1\ldots$, either finite or infinite. Each $a_t$ is an atomic action taken by either $p_0$ or $p_1$. Each $S_t$ is a state of the n-process system—a ordered quintuple $(k, PC, q, t, op)$, where $k$, $PC$, $q$, $t$, and $op$ are arrays indexed by $\{1, \ldots, n\}$. A schedule $\beta$ of the execution $\alpha$ is the sequence $S_0S_1\ldots$. Note that $\alpha$ is uniquely defined by $\beta$. In the following mutual exclusion correctness proof of the Peterson-Fischer 2-process algorithm, we will consider all possible schedules $\beta$.

The following conventions will be used when discussing states of the system: For a state $S_t = ((k_1, \ldots, k_n), (PC_1, \ldots, PC_n), (q_1, \ldots, q_n), (t_1, \ldots, t_n), (op_1, \ldots, op_n))$, $S_t.k[i] = k_i$, $S_t.PC[i] = PC_i$, $S_t.q[i] = q_i$, $S_t.t[i] = t_i$, and $S_t.op[i] = op_i$. Furthermore, if $PC_i = (a, b)$, then first($S_t.PC[i]$) = $a$. Also, $S_t = S_{t_2}$ iff all elements of $S_{t_1}$ are equal to the corresponding six elements of $S_{t_2}$. If an element's value is said to be $*$, then its value does not matter (i.e., it may take on any value without affecting the truth of the statement).
Atomic Step \((0, j), \forall 1 \leq j \leq n:\)
\[
\text{op}[i] \leftarrow \text{op}[i] \setminus \{j\}
\]
if \(\text{level}(q[j]) \geq k[i]\)
then \(t[i] \leftarrow q[j]; \ PC[i] \leftarrow (1, 0)\)
else if \(\text{op}[i] \neq \emptyset\)
then for some \(m \in \text{op}[i], \ PC[i] \leftarrow (0, m)\)
else \(PC[i] \leftarrow (1, 0)\)

Atomic Step \((1, 0):\)
if \(\text{level}(t[i]) \neq k[i]\)
then \(q[i] \leftarrow (k[i], T)\)
else \(q[i] \leftarrow (k[i], \text{bit}(i, k[i]) \oplus \text{flag}(t[i]))\)
\(t[i] \leftarrow (0, F)\)
\(\text{op}[i] \leftarrow \text{opponents}(i, k[i])\)
for some \(m \in \text{op}[i], \ PC[i] \leftarrow (2, m)\)

Atomic Step \((2, j), \forall 1 \leq j \leq n:\)
\[
\text{op}[i] \leftarrow \text{op}[i] \setminus \{j\}
\]
if \(\text{level}(q[j]) \geq k[i]\)
then \(t[i] \leftarrow q[j]; \ PC[i] \leftarrow (3, 0)\)
else if \(\text{op}[i] \neq \emptyset\)
then for some \(m \in \text{op}[i], \ PC[i] \leftarrow (2, m)\)
else \(PC[i] \leftarrow (3, 0)\)

Atomic Step \((3, 0):\)
if \(\text{level}(t[i]) = k[i]\)
then \(q[i] \leftarrow (k[i], \text{bit}(i, k[i]) \oplus \text{flag}(t[i]))\)
else \(q[i] \leftarrow q[i]\)
\(t[i] \leftarrow (0, F)\)
\(\text{op}[i] \leftarrow \text{opponents}(i, k[i])\)
for some \(m \in \text{op}[i], \ PC[i] \leftarrow (4, m)\)

Atomic Step \((4, j), \forall 1 \leq j \leq n:\)
\[
\text{op}[i] \leftarrow \text{op}[i] \setminus \{j\}
\]
if \(\text{level}(q[j]) < k[i]\) and \(\text{op}[i] \neq \emptyset\)
then for some \(m \in \text{op}[i], \ PC[i] \leftarrow (4, m)\)
else if \(\text{level}(q[j]) < k[i]\) or \((\text{level}(q[j]) = k[i] \text{ and } \text{bit}(i, k[i]) \oplus \text{flag}(q[j]) \neq \text{flag}(q[i]))\)
then if \(k[i] = \lceil \lg n \rceil\)
then \(k[i] \leftarrow k[i] + 1; \ PC[i] \leftarrow (5, 0)\)
else \(k[i] \leftarrow k[i] + 1; \ \text{op}[i] = \text{opponents}(i, k[i]); \ \text{for some } m \in \text{op}[i], \ PC[i] \leftarrow (0, m)\)
else \(\text{op}[i] = \text{opponents}(i, k[i]); \ \text{for some } m \in \text{op}[i], \ PC[i] \leftarrow (4, m)\)

Atomic Step \((5, 0):\)
\(q[i] \leftarrow (0, F)\)
\(t[i] \leftarrow (0, F)\)
\(k[i] \leftarrow 1\)
\(\text{op}[i] \leftarrow \text{opponents}(i, 1)\)
for some \(m \in \text{op}[i], \ PC[i] \leftarrow (0, m)\)

Figure 3-3: The Atomic Steps of the \(n\)-process algorithm.
Shared variables:

- \( q \): an array indexed by \( \{1, \ldots, n\} \) of pairs (level, flag), where level is an integer and flag takes on values in \( \{T, F\} \). Initially, \( q[i] = (0, F) \) for all \( i \). Variable \( q[i] \) is written by \( p_i \) and read by all.

Local variables:

- \( t \): an array indexed by \( \{1, \ldots, n\} \) of pairs (level, flag), where level is an integer and flag takes on values in \( \{T, F\} \). Initially, \( t[i] = (0, F) \) for all \( i \). Variable \( q[i] \) is written and read only by \( p_i \).

- \( PC \): an array indexed by \( \{1, \ldots, n\} \) of pairs \((a, b)\), where \( a \in \{0, 1, \ldots, 5\} \) and \( b \in \{0, \ldots, n\} \). Initially, \( PC[i] = (0, m) \), where \( m \) is the only member of opponents\((i, 1)\). Variable \( PC[i] \) is written only by \( p_i \) and never read.

- \( k \): an array indexed by \( \{1, \ldots, n\} \) of values from \( \{1, \ldots, \lfloor \log n \rfloor + 1\} \), initially all 1, where \( k[i] \) is written and read only by \( p_i \).

- \( op \): an array indexed by \( \{1, \ldots, n\} \) of subsets of \( \{1, \ldots, n\} \). Initially, \( op[i] = \text{opponents}(i, 1) \). Variable \( op[i] \) is written and read only by \( p_i \).

Code for \( p_i \): At every step, execute Atomic Step \( PC[i] \). The actual Atomic Steps are shown in Figure 3-3.

Definition:

- \( p_i \) is in the Critical region iff \( PC[i] = (5, 0) \) iff \( k[i] = \lfloor \log n \rfloor + 1. \)

Figure 3-4: The Peterson-Fischer \( n \)-process mutual exclusion algorithm shown as atomic steps. The flow of control is defined by the value of \( PC[i] \).
3.3 The n-process algorithm satisfies mutual exclusion

First, we make some useful statements about the opponent function.

Lemma 3.1 The following statements are equivalent, for all $0 \leq k \leq \lceil \log n \rceil$:

\[ j \in \text{opponents}(i, k) \]
\[ i \in \text{opponents}(j, k) \]
\[ \text{opponents}(i, k) = \{j\} \cup \bigcup_{l=1}^{k-1} \text{opponents}(j, l) \]
\[ \text{opponents}(j, k) = \{i\} \cup \bigcup_{l=1}^{k-1} \text{opponents}(i, l) \]

Proof: True by definition of the opponent function. \hfill \Box

Now, we relate the level field of a shared variable $q[i]$ with $k[i]$ during some state with the following Lemma.

Lemma 3.2 For any process $p_i$, $k[i] = \text{level}(q[i]) + 1$ iff $\text{first}(PC[i]) \in \{0, 1, 5\}$. Otherwise, $k[i] = \text{level}(q[i])$. I.e.:

\[ \forall t \geq 0 \forall 1 \leq i \leq n [S_t.k[i] = \text{level}(S_t.q[i]) + 1 \iff \text{first}(S_t.PC[i]) \in \{0, 1, 5\}] \]
\[ \forall t \geq 0 \forall 1 \leq i \leq n [S_t.k[i] = \text{level}(S_t.q[i]) \iff \text{first}(S_t.PC[i]) \in \{2, 3, 4\}] \]

Proof: Examination of algorithm. \hfill \Box

Lemma 3.3 For any possible schedule $\beta$ of the Peterson-Fischer n-process mutual exclusion algorithm,

\[ \forall 0 \leq k \leq \lceil \log n \rceil \forall t \geq 0 \forall 1 \leq i \leq n \forall j \in \text{opponents}(i, k)[S_t.k[i] \leq k \lor S_t.k[j] \leq k]. \]

Proof: By induction on $k$. Basis step: $k = 0$. Since $\forall 1 \leq i \leq n [j \in \text{opponents}(i, k) = \emptyset]$, the basis step is satisfied. Inductive step. Assume

\[ \forall 0 \leq k' \leq k \forall t \geq 0 \forall 1 \leq i \leq n \forall j \in \text{opponents}(i, k')[S_t.k[i] \leq k' \lor S_t.k[j] \leq k']. \]

Show

\[ \forall t \geq 0 \forall 1 \leq i \leq n \forall j \in \text{opponents}(i, k+1)[S_t.k[i] \leq k + 1 \lor S_t.k[j] \leq k + 1]. \]
We proceed by contradiction. Assume
\[ \exists t \geq 0 \exists 1 \leq i \leq n \exists j \in \text{opponents}(i, k + 1) \left( S_{i', k}[i] > k + 1 \land S_{i', k}[j] > k + 1 \right) \]
and fix \( t, i, \) and \( j \) with this property.

Let \( t_i \) be the greatest value less than \( t \) such that \( S_{t_i - 1, k}[i] = k \). Let \( t_j \) be the greatest value less than \( t \) such that \( S_{t_j - 1, k}[j] = k \). Assume, without loss of generality, that \( t_i < t_j \).

**Claim 3.4** The following statements are true for all \( S_{t'} \) where \( t_j \leq t' \leq t \):

1. \( S_{t', k}[i] \geq k + 1 \land S_{t', k}[j] \geq k + 1 \)
2. \( \text{level}(S_{t', q[i]}) \geq k \land \text{level}(S_{t', q[j]}) \geq k \)
3. \( m \in \text{opponents}(i, k + 1) \land m \neq j \implies \text{level}(S_{t', q[m]}) \leq k \)
4. \( m \in \text{opponents}(j, k + 1) \land m \neq i \implies \text{level}(S_{t', q[m]}) \leq k \)

**Proof:** From the definition of \( t_j \) and \( t \), we know that \( S_{t_j - 1, k}[j] = k \) and
\[ \forall t_j \leq t' \leq t \left[ S_{t', k}[i] \geq k + 1 \land S_{t', k}[j] \geq k + 1 \right]. \]
This is Statement 1 of the Claim. Then, by Lemma 3.2,
\[ \forall t_j \leq t' \leq t \left[ \text{level}(S_{t', q[i]}) \geq k \land \text{level}(S_{t', k}[j]) \geq k \right]. \]
This is Statement 2 of the Claim. From the inductive hypothesis, we know that
\[ \forall 1 \leq k' \leq k \forall t_j \leq t' \leq t \left[ \forall m \in \text{opponents}(i, k') \left[ S_{t', k}[m] \leq k \right] \land \forall m \in \text{opponents}(j, k') \left[ S_{t', k}[m] \leq k \right] \right]. \]
Since \( j \in \text{opponents}(i, k + 1) \), Lemma 3.1 tells us that
\[ \text{opponents}(i, k + 1) = \{ j \} \cup \bigcup_{l=1}^{k} \text{opponents}(j, l) \]
and
\[ \text{opponents}(j, k + 1) = \{ i \} \cup \bigcup_{l=1}^{k} \text{opponents}(i, l). \]
Thus,
\[ \forall t_j \leq t' \leq t \left[ m \in \text{opponents}(i, k + 1) \land S_{t', k}[m] \geq k + 1 \implies m = j \right] \]

22
and
\[ \forall t_j \leq t \leq t \left[ m \in \text{opponents}(j, k + 1) \land S_{t', k}[m] \geq k + 1 \implies m = i \right] \]

By Lemma 3.2,
\[ \forall t_j \leq t \leq t \left[ m \in \text{opponents}(i, k + 1) \land m \neq j \implies \text{level}(S_{t'}q[m]) \leq k \right] \]
and
\[ \forall t_j \leq t \leq t \left[ m \in \text{opponents}(j, k + 1) \land m \neq i \implies \text{level}(S_{t'}q[m]) \leq k \right] \]

These are Statements 3 and 4 of the Claim.

At this point, we will establish a mapping between states of the n-process system in the interval \([S_t, S_{t'}]\) and states of a 2-process system as defined in Chapter 2. Note that, since \(i \in \text{opponents}(j, k + 1)\) (and thus \(j \in \text{opponents}(i, k + 1)\)), it follows that \(\text{bit}(i, k + 1) = -\text{bit}(j, k + 1)\). For \(r \in \{i, j\}\), let \(b(r) = \text{bit}(r, k + 1)\). The general strategy will be, for \(r \in \{i, j\}\), to have \(p_r\) of the n-process system play the role of \(p_{b(r)}\) of the 2-process system, where \(k[r] = k + 1\) will correspond to the Trying region of the 2-process code and \(k[r] > k + 1\) will correspond to the Critical region of the 2-process code. (Recall from Claim 3.4 that \(k[r] \geq k + 1\).)

After the mapping is defined, we will show that it satisfies the following three properties:

**Property 1:** \(S_{t_j}\) maps to a reachable state of the 2-process system.

**Property 2:** \(S_t\) does not map to a reachable state of the 2-process system.

**Property 3:** For any \(t_j \leq t' < t\), if \(S_{t'}\) maps to a reachable state of the 2-process system, then \(S_{t'+1}\) does, also.

Since these three Properties cannot all be true, we may then conclude that the assumption that such a \(S_t\) existed was flawed, and the proof will be established.

Before we define the mapping, we first must define a pair of constants, \(C_i\) and \(C_j\). Conceptually, the purposes of \(C_i\) and \(C_j\) are to keep track of the values of \(q[i]\) and \(q[j]\) when their associated processes “entered the Critical region” of the 2-process system. In other words, for \(r \in \{i, j\}\), \(q[r]\) may change in the interval \([S_{t_j}, S_t]\) after the n-process action that will be analagous to the transition of process \(p_{b(r)}\) of the 2-process system to the Critical region, but we want to define the mapping to act as if it is static. Note that for \(r \in \{i, j\}\), there can be
only one \( t_j \leq t' < t \) such that \( S_{t'}.k[r] = k + 2 \) and \( a_{t'} \) is Atomic Step \((1,0)\) of \( p_r \). This is the first alteration of \( q[r] \) that we do not want to reflect in the corresponding 2-process state. For \( r \in \{i, j\} \), define

\[
C_r = \begin{cases} 
\text{flag}(S_{t'}.q[r]) & \text{if such a } t' \text{ exists} \\
T & \text{otherwise}
\end{cases}
\]

Thus, we are "saving" the value of \( q[r] \) immediately preceding that action.

Now, let us define the mapping. Remember that \( \text{first}(PC) \) denote the first element of \( PC \)'s ordered pair. Also, remember that \( a_{t'} \) denotes the Atomic Step between \( S_{t'} \) and \( S_{t'+1} \). Let

\[
\text{opp}(i) = j \\
\text{opp}(j) = i
\]

Then,

\[
f(S) = s,
\]

where, for \( r \in \{i, j\} \),

\[
s_{.PC}[b(r)] = \begin{cases} 
5 & \text{if } S.k[r] \neq k + 1 \text{ (i.e., } S.k[r] > k + 1) \\
\text{first}(S.PC[r]) & \text{if } S.k[r] = k + 1 \\
& \text{and } (\text{opp}(r) \in S.\text{op}[r] \text{ or first}(S.PC[r]) \in \{1, 3, 5\}) \\
\text{first}(S.PC[r]) + 1 & \text{if } S.k[r] = k + 1 \\
& \text{and } \text{opp}(r) \notin S.\text{op}[r] \text{ and first}(S.PC[r]) \in \{0, 2, 4\}
\end{cases}
\]

\[
s_{.q}[b(r)] = \begin{cases} 
\text{nil} & \text{if level}(S.q[r]) \leq k \text{ (i.e., } \text{level}(S.q[r]) = k) \\
\text{flag}(S.q[r]) & \text{if level}(S.q[r]) = k + 1 \\
C_r & \text{if level}(S.q[r]) > k + 1
\end{cases}
\]

\[
s_{.t}[b(r)] = \begin{cases} 
\text{nil} & \text{if level}(S.t[r]) \leq k \text{ (i.e., } \text{level}(S.t[r]) = k) \\
\text{flag}(S.t[r]) & \text{if } S.k[r] = \text{level}(S.t[r]) = k + 1 \\
\text{nil} & \text{if } S.k[r] > k + 1
\end{cases}
\]

Claim 3.5 The mapping \( f \) satisfies Property 1, Property 2, and Property 3, described above.

Proof:
Property 1: $S_{t_j}$ maps to a reachable state of the 2-process system. Let $s_{t_j} = f(S_{t_j})$. Since $t_j$ was defined to be the greatest value less than $t$ such that $S_{t_j-1}.k[j] = k$, we know that $a_{t_j-1}$ (the atomic action between $S_{t_j-1}$ and $S_{t_j}$) was Atomic Step (4, $m$) of Figure 3-3 for some $m$. Also, since $S_{t_j}.k[j] = k + 1$ and $k + 1 \leq \lfloor \lg n \rfloor$, we know that $p_j$ took during action $a_{t_j-1}$ the following branch of Atomic Step (4, $j$):

$$\begin{align*}
k[j] &\leftarrow k + 1; \text{opp}[i] = \text{opponents}(i, k + 1); \text{for some } m \in \text{op}[i], PC[i] \leftarrow (0, m)
\end{align*}$$

Knowing this, we will now determine properties of each of the six elements of $s_{t_j}$ and show that $s_{t_j}$ is a possible starting state of the 2-process algorithm (and thus reachable).

- **$s_{t_j}.PC[0]$**: The 2-process system imposes no restrictions on the initial value of $s_{t_j}.PC[0]$, so no matter what $s_{t_j}.PC[0]$ is, it meets the requirements for a 2-process starting state.

- **$s_{t_j}.q[0]$**: We know that $s_{t_j}.k[i] \geq k + 1$, and thus level($s_{t_j}.q[i]$) $\geq k$. Now, $s_{t_j}.q[0] = \text{nil}$ iff level($s_{t_j}.q[i]$) $= k$ (and thus $s_{t_j}.k[i] = k + 1$). Then, by Lemma 3.2, $s_{t_j}.q[0] = \text{nil}$ iff first($s_{t_j}.PC[i]$) $\in \{0, 1, 5\}$. Since $s_{t_j}.k[i] = k + 1$ and $k + 1 \leq \lfloor \lg n \rfloor$, it follows that first($s_{t_j}.PC[i]$) $\neq 5$. So, $s_{t_j}.q[0] = \text{nil}$ iff first($s_{t_j}.PC[i]$) $\leq 1$. Thus, $s_{t_j}.q[0] = \text{nil}$ iff $s_{t_j}.PC[0] \leq 1$. This is precisely the requirement imposed on the initial state of $q[0]$ in the 2-process system. Thus, $s_{t_j}.q[0]$ meets the requirements for a 2-process starting state.

- **$s_{t_j}.t[0]$**: The 2-process system imposes no restrictions on the initial value of $t[0]$, so no matter what $s_{t_j}.t[0]$ is, it meets the requirements for a 2-process starting state.

- **$s_{t_j}.PC[1]$**: We know that first($s_{t_j}.PC[j]$) $= 0$ and $j \in s_{t_j}.op[i]$ (since $j \in \text{opponents}(i, k + 1)$). So, $s_{t_j}.PC[1] = 0$, and thus meets the requirements for a 2-process starting state.

- **$s_{t_j}.q[1]$**: Since first($s_{t_j}.PC[j]$) $= 0$ and $s_{t_j}.k[j] = k + 1$, we know by Lemma 3.2 that level($s_{t_j}.q[j]$) $= k$. So, $s_{t_j}.q[1] = \text{nil}$ and thus meets the requirements for a 2-process starting state.

- **$s_{t_j}.t[1]$**: Since $t[i]$ gets values exclusively from $q[i]$ and level($s_{t_j}.q[j]$) $= k$, we know that level($s_{t_j}.t[j]$) $\leq k$. So, $s_{t_j}.t[1] = \text{nil}$ and thus meets the requirements for a 2-process starting state.
So, \( f(S_t) \) is a valid starting state of the 2-process system, and therefore is reachable.

**Property 2:** \( S_t \) does not map to a reachable state of the 2-process system. By our original “contradiction” assumption, \( S_t.k[i] > k + 1 \) and \( S_t.k[j] > k + 1 \). Thus, if \( s_t = f(S_t) \), then \( s_t.PC[0] = 5 \) and \( s_t.PC[1] = 5 \). However, by Theorem 2.7, this is not a reachable state of the 2-process algorithm.

**Property 3:** For any \( t_j < t' < t \), if \( S_{t'} \) maps to a reachable state of the 2-process system, then \( S_{t'+1} \) does, also. Let \( s_{t'} = f(S_{t'}) \) and let \( s_{t'+1} = f(S_{t'+1}) \). Let \( a_{t'} \) be the atomic action (i.e., the Atomic Step of some process) between \( S_{t'} \) and \( S_{t'-1} \). Let “\( x \) is unchanged” denote the fact that \( S_{t'}.x = S_{t'+1}.x \). Note that for some state \( S \), the only items used in the calculation of \( f(S) \) are, for \( r \in \{i, j\} \), \( S.k[r] \), \( S.q[r] \), \( S.t[r] \), and \( S.op[r] \).

Also note from the code in Figure 3-3 that, for \( r \in \{i, j\} \):

- \( k[r] \) can be changed only by READs by \( p_r \)
- \( q[r] \) can be changed only by WRITEs by \( p_r \)
- \( op[r], PC[r], \) and \( t[r] \) can be changed only by READs by \( p_r \) and WRITEs by \( p_r \)

Now, we shall examine all possible cases for \( a_{t'} \). An outline for all of the cases is shown in Figure 3-5. It refers to the numbers below, where all of the cases are actually analyzed.

1. \( a_{t'} \) is a READ or WRITE by \( p_m \), where \( m \neq i \) and \( m \neq j \):

   In this case, \( s_{t'+1} = s_{t'} \), and thus \( s_{t'+1} \) is reachable.

2. For some \( r \in \{i, j\} \), \( a_{t'} \) is a WRITE by \( p_r \) and \( \text{level}(S_{t'}.q[r]) > k + 1 \):

   In this case, \( S_{t'}.k[r] \) must be \( k + 1 \). Furthermore, \( \text{level}(S_{t'+1}.q[r]) = k + 1 \) and \( S_{t'+1}.k[r] \) \( > k + 1 \). So, \( s_{t'}.PC[b(r)] = s_{t'+1}.PC[b(r)] = 5 \), \( s_{t'}.q[b(r)] = s_{t'+1}.q[b(r)] = C_r \), and \( s_{t'}.t[b(r)] = s_{t'+1}.t[b(r)] = \text{nil} \). Therefore \( s_{t'+1} = s_{t'} \), and thus \( s_{t'+1} \) is reachable.

3. For some \( r \in \{i, j\} \), \( a_{t'} \) is a WRITE by \( p_r \), \( \text{level}(S_{t'}.q[r]) = k + 1 \), and \( S_{t'}.k[r] \) \( > k + 1 \):

   By Lemma 3.2, \( S_{t'}.k[r] = k + 2 \) and \( \text{first}(S_{t'}.PC[r]) \) \( \in \{0, 1, 5\} \). Since \( a_{t'} \) is a WRITE, \( \text{first}(S_{t'}.PC[r]) \neq 0 \). If \( \text{first}(S_{t'}.PC[r]) = 5 \), then \( S_{t'+1}.k[r] = 1 \). However, \( S_{t'+1}.k[r] = k + 1 \). So, \( \text{first}(S_{t'}.PC[r]) = 1 \), and \( a_{t'} \) is Atomic Step \((1, 0)\) of \( p_r \). So,
\* \(a_{\nu}\) is an action by \(p_m\), where \(m \neq i\) and \(m \neq j\) (Case 1)

\* \(a_{\nu}\) is an action by \(p_r\), where \(r \in \{i, j\}\)

\* \(a_{\nu}\) is a WRITE
  \(S_{\nu}.k[r] \neq k + 1\) \((i.e., > k + 1)\)
  \* \(\text{level}(S_{\nu}.q[r]) > k + 1\) (Case 2)
  \* \(\text{level}(S_{\nu}.q[r]) = k + 1\) (Case 3)
  \(S_{\nu}.k[r] = k + 1\)
  \* \(S_{\nu}.PC[r] \in \{0, 2, 4\}\) (Case 4)
  \* \(S_{\nu}.PC[r] = 5\) (Case 5)
  \* \(S_{\nu}.PC[r] = 1\) (Case 6)
  \* \(S_{\nu}.PC[r] = 3\) (Case 7)

\* \(a_{\nu}\) is a READ
  \(S_{\nu}.k[r] \neq k + 1\) \((i.e., > k + 1)\) (Case 8)
  \(S_{\nu}.k[r] = k + 1\)
  \* \(S_{\nu}.PC[r] \in \{1, 3, 5\}\) (Case 9)
  \* \(S_{\nu}.PC[r] = 0\)
    \* \(m \neq \text{opp}(r)\) (Case 10)
    \* \(m = \text{opp}(r)\) (Case 11)
  \* \(S_{\nu}.PC[r] = 2\)
    \* \(m \neq \text{opp}(r)\) (Case 12)
    \* \(m = \text{opp}(r)\) (Case 13)
  \* \(S_{\nu}.PC[r] = 4\)
    \* \(m \neq \text{opp}(r)\) (Case 14)
    \* \(m = \text{opp}(r)\) (Case 15)

Figure 3-5: Possible cases for \(a_{\nu}\).
level($S_{\nu+1}.q[r]$) = $k+2$. Thus, $s_{\nu}.q[b(r)] = level(S_{\nu}.q[r])$ and $s_{\nu+1}.q[b(r)] = C_r$. But, by the definition of $C_r$, $C_r = level(S_{\nu}.q[r])$. So, $s_{\nu}.q[b(r)] = s_{\nu+1}.q[b(r)]$. Also, since $S_{\nu}.k[r] > k + 1$, $s_{\nu}.PC[b(r)] = s_{\nu+1}.PC[b(r)] = 5$ and $s_{\nu}.t[b(r)] = s_{\nu+1}.t[b(r)] = \text{nil}$. Therefore $s_{\nu+1} = s_{\nu}$, and thus $s_{\nu+1}$ is reachable.

4. $a_{\nu}$ is a WRITE by $p_m$ and first($S_{\nu}.PC[m]$) $\in \{0, 2, 4\}$, $1 \leq m \leq n$:

No such $t'$ exists, because if $a_{\nu}$ is a WRITE by $p_m$, then first($S_{\nu}.PC[m]$) $\in \{1, 3, 5\}$.

5. For some $r \in \{i, j\}$, $a_{\nu}$ is a WRITE by $p_r$ and first($S_{\nu}.PC[r]$) $= 5$:

No such $t'$ exists, because then $S_{\nu+1}.k[r] = 1 < k + 1$, but we know that $S_{\nu+1}.k[r]$ must be $\geq k + 1$.

6. For some $r \in \{i, j\}$, $a_{\nu}$ is a WRITE by $p_r$, $S_{\nu}.k[r] = k + 1$, and first($S_{\nu}.PC[r]$) $= 1$:

In this case, $a_{\nu}$ is Atomic Step (1, 0) of $p_r$. We know that first($S_{\nu+1}.PC[r]$) $= 2$ and opp($r$) $\in S_{\nu+1}.op[r]$. So, $s_{\nu}.PC[b(r)] = 1$ and $s_{\nu+1}.PC[b(r)] = 2$. Also, $s_{\nu+1}.t[b(r)] = \text{nil}$. There are two cases for $S_{\nu+1}.q[r]$:

(a) $level(S_{\nu}.t[r]) \neq k + 1$. In this case, $S_{\nu+1}.q[r] = (k + 1, T)$. So, $s_{\nu}.t[b(r)] = \text{nil}$ and $s_{\nu+1}.q[b(r)] = T$. Thus, Atomic Step 1 of $p_{b(r)}$ of the 2-process system after $s_{\nu}$ will yield $s_{\nu+1}$. Therefore, $s_{\nu+1}$ is reachable.

(b) $level(S_{\nu}.t[r]) = k + 1$. In this case, $S_{\nu+1}.q[r] = (k + 1, \text{flag}(S_{\nu}.t[r]))$. So, $s_{\nu}.t[b(r)] \neq \text{nil}$ and $s_{\nu+1}.q[b(r)] = s_{\nu}.t[b(r)]$. Thus, Atomic Step 1 of $p_{b(r)}$ of the 2-process system after $s_{\nu}$ will yield $s_{\nu+1}$. Therefore, $s_{\nu+1}$ is reachable.

7. For some $r \in \{i, j\}$, $a_{\nu}$ is a WRITE by $p_r$, $S_{\nu}.k[r] = k + 1$, and first($S_{\nu}.PC[r]$) $= 3$:

In this case, $a_{\nu}$ is Atomic Step (3, 0) of $p_r$. We know that first($S_{\nu+1}.PC[r]$) $= 4$ and opp($r$) $\in S_{\nu+1}.op[r]$. So, $s_{\nu}.PC[b(r)] = 3$ and $s_{\nu+1}.PC[b(r)] = 4$. Also, $s_{\nu+1}.t[b(r)] = \text{nil}$. There are two cases for $S_{\nu+1}.q[r]$:

(a) $level(S_{\nu}.t[r]) \neq k + 1$. In this case, $S_{\nu+1}.q[r] = S_{\nu}.q[r]$. So, $s_{\nu}.t[b(r)] = \text{nil}$ and $s_{\nu+1}.q[b(r)] = s_{\nu}.q[b(r)]$. Thus, Atomic Step 3 of $p_{b(r)}$ of the 2-process system after $s_{\nu}$ will yield $s_{\nu+1}$. Therefore, $s_{\nu+1}$ is reachable.

(b) $level(S_{\nu}.t[r]) = k + 1$. In this case, $S_{\nu+1}.q[r] = (k + 1, \text{flag}(S_{\nu}.t[r]))$. So, $s_{\nu}.t[b(r)] \neq \text{nil}$ and $s_{\nu+1}.q[b(r)] = s_{\nu}.t[b(r)]$. Thus, Atomic Step 3 of $p_{b(r)}$
of the 2-process system after $s_t$ will yield $s_{t+1}$. Therefore, $s_{t+1}$ is reachable.

8. For some $r \in \{i, j\}$, $a_t$ is a READ by $p_r$ and $S_{t}.k[r] \neq k + 1$ (i.e., $> k + 1$):

In this case, $S_{t+1}.k[r]$ must also be $\neq k + 1$. So, $s_{t}.PC[b(r)] = s_{t+1}.PC[b(r)] = 5$ and $s_{t}.t[b(r)] = s_{t+1}.t[b(r)] = \text{nil}$. Also, since $a_t$ is a READ, $s_{t}.q[b(r)] = s_{t+1}.q[b(r)]$. Therefore $s_{t+1} = s_t$, and thus $s_{t+1}$ is reachable.

9. $a_t$ is a READ by $p_m$ and $\text{first}(S_{t}.PC[m]) \in \{1, 3, 5\}$, $1 \leq m \leq n$:

No such $t'$ exists, because if $a_t$ is a READ by $p_m$, then $\text{first}(S_{t}.PC[m]) \in \{0, 2, 4\}$.

10. For some $r \in \{i, j\}$, $a_t$ is a READ by $p_r$ of $q[m]$, where $m \neq \text{opp}(r)$, $S_{t}.k[r] = k + 1$, and $\text{first}(S_{t}.PC[r]) = 0$:

In this case, $a_t$ is Atomic Step $(0, m)$ of $p_r$. Since $S_{t}.k[r] = k + 1$, $m \in \text{opponents}(r, k + 1)$. So, since $m \neq \text{opp}(r)$, it follows from Claim 3.4 that $\text{level}(q[m]) \leq k < S_{t}.k[r]$. Thus, the else branch of Atomic Step $(0, m)$ is taken. So, $t[r]$ and $q[r]$ are unchanged, and $\text{opp}(r) \in S_{t}.op[r] \iff \text{opp}(r) \in S_{t+1}.op[r]$. Examination of Atomic Step $(0, m)$ reveals that there are two cases for $S_{t+1}.PC[r]$:

(a) $S_{t}.op[r] \neq \emptyset$. In this case, $\text{first}(PC[r])$ is unchanged and $\text{opp}(r) \in S_{t}.op[r]$ iff $\text{opp}(r) \in S_{t+1}.op[r]$. Thus, $s_t = s_{t+1}$, and therefore $s_{t+1}$ is reachable.

(b) $S_{t}.op[r] = \emptyset$. In this case, $\text{opp}(r) \notin S_{t}.op[r]$, $\text{first}(S_{t}.PC[r]) = 0$, and $\text{first}(S_{t+1}.PC[r]) = 1$. So, $s_{t}.PC[b(r)] = s_{t+1}.PC[b(\text{opp}(r))] = 1$. Thus, $s_t = s_{t+1}$, and therefore $s_{t+1}$ is reachable.

11. For some $r \in \{i, j\}$, $a_t$ is a READ by $p_r$ of $q[\text{opp}(r)]$, $S_{t}.k[r] = k + 1$, and $\text{first}(S_{t}.PC[r]) = 0$:

In this case, $a_t$ is Atomic Step $(0, \text{opp}(r))$ of $p_r$. We know that $\text{opp}(r) \in S_{t}.op[r]$, $\text{opp}(r) \in S_{t+1}.op[r]$, and $\text{first}(S_{t+1}.PC[r]) \in \{0, 1\}$. So, $s_{t}.PC[b(r)] = 0$ and $s_{t+1}.PC[b(r)] = 1$. Also, $q[r]$ is unchanged, so $s_{t}.q[b(r)] = s_{t}.q[b(\text{opp}(r))]$. There are two cases for $S_{t+1}.t[r]$:

(a) $\text{level}(S_{t}.q[\text{opp}(r)]) \geq k + 1$. In this case, $S_{t+1}.t[r] = S_{t+1}.q[\text{opp}(r)]$. So, $s_{t}.q[b(\text{opp}(r))] \neq \text{nil}$ and $s_{t+1}.t[b(r)] = s_{t+1}.q[b(\text{opp}(r))]$. Thus, Atomic Step 0 of $p_{b(r)}$ of the 2-process system after $s_t$ will yield $s_{t+1}$. Therefore, $s_{t+1}$ is reachable.
(b) \( \text{level}(S_{\nu}.q[\text{opp}(r)]) \leq k \). In this case, \( s_{\nu}.q[b(\text{opp}(r))] = \text{nil} \) and \( s_{\nu}.t[b(r)] = s_{\nu+1}.t[b(r)] \). Thus, Atomic Step 0 of \( p_b(r) \) of the 2-process system after \( s_{\nu} \) will yield \( s_{\nu+1} \). Therefore, \( s_{\nu+1} \) is reachable.

12. For some \( r \in \{i, j\} \), \( a_{\nu} \) is a READ by \( p_r \) of \( q[m] \), where \( m \neq \text{opp}(r) \), \( S_{\nu}.k[r] = k + 1 \), and \( \text{first}(S_{\nu}.PC[r]) = 2 \):

This case is completely analogous to Case 10, substituting Atomic Step \((2, m)\) for Atomic Step \((0, m)\).

13. For some \( r \in \{i, j\} \), \( a_{\nu} \) is a READ by \( p_r \) of \( q[\text{opp}(r)] \), \( S_{\nu}.k[r] = k + 1 \), and \( \text{first}(S_{\nu}.PC[r]) = 2 \):

This case is completely analogous to Case 11, but relating Atomic Step \((2, \text{opp}(r))\) of \( p_r \) in the \( n \)-process system to to Atomic Step 2 of \( p_b(r) \) in the 2-process system, instead.

14. For some \( r \in \{i, j\} \), \( a_{\nu} \) is a READ by \( p_r \) of \( q[m] \), where \( m \neq \text{opp}(r) \), \( S_{\nu}.k[r] = k + 1 \), and \( \text{first}(S_{\nu}.PC[r]) = 4 \):

In this case, \( a_{\nu} \) is Atomic Step \((4, m)\) of \( p_r \). We know that \( t[r] \), and \( q[r] \) are unchanged, and \( \text{opp}(r) \in S_{\nu}.\text{op}[r] \iff \text{opp}(r) \in S_{\nu+1}.\text{op}[r] \). Since \( S_{\nu}.k[r] = k + 1 \), \( m \in \text{opponents}(r, k + 1) \). So, since \( m \neq \text{opp}(r) \), it follows from Claim 3.4 that \( \text{level}(q[m]) \leq k < S_k.k[r] \). Examination of Atomic Step \((4, m)\) reveals that there are two cases for \( S_{\nu+1}.PC[r] \):

(a) \( S_{\nu}.\text{op}[r] \neq \emptyset \). This corresponds to the first \textbf{then} in Atomic Step \((4, j)\). In this case, \( \text{first}(PC[r]) \) and \( \text{opp}(r) \in S_{\nu}.\text{op}[r] \iff \text{opp}(r) \in S_{\nu+1}.\text{op}[r] \). Thus, \( s_{\nu} = s_{\nu+1} \), and therefore \( s_{\nu+1} \) is reachable.

(b) \( S_{\nu}.\text{op}[r] = \emptyset \). In this case, \( \text{opp}(r) \notin S_{\nu}.\text{op}[r] \). Also, \( S_{\nu+1}.k[r] = S_{\nu}.k[r] + 1 = k + 2 > k + 1 \). So, \( s_{\nu}.PC[b(r)] = s_{\nu+1}.PC[b(r)] = 5 \). Thus, \( s_{\nu} = s_{\nu+1} \), and therefore \( s_{\nu+1} \) is reachable.

15. For some \( r \in \{i, j\} \), \( a_{\nu} \) is a READ by \( p_r \) of \( q[\text{opp}(r)] \), \( S_{\nu}.k[r] = k + 1 \), and \( \text{first}(S_{\nu}.PC[r]) = 4 \):

In this case, \( a_{\nu} \) is Atomic Step \((4, \text{opp}(r))\) of \( p_r \). We know that \( \text{opp}(r) \in S_{\nu}.\text{op}[r] \) and that \( q[r] \) and \( t[r] \) are unchanged. There are three cases for \( S_{\nu+1}.\text{op}[r] \) and
Thus, by Claim 3.5, f satisfies the property that for any \( t_j \leq t' < t \), if \( S_{t'} \) maps to a reachable state of the 2-process system, then \( S_{t'+1} \) does, also. However, we also showed that \( f(S_t) \) is a reachable state of the 2-process algorithm, but \( f(S_{t'}) \) is not. This is a contradiction. Thus, our original assumption that such a \( S_t \) existed was flawed, and the proof of Lemma 3.3 is established.

\[ \textbf{Theorem 3.6} \quad \text{The Peterson-Fischer n-process mutual exclusion algorithm satisfies mutual exclusion.} \]

\textbf{Proof:} Consider any two processes, \( p_i \) and \( p_j \) at any reachable state \( S_t \). There exists a \( k \) such that \( 1 \leq k \leq \lceil \lg n \rceil \) and \( j \in \text{opponents}(i, k) \). Then, from Lemma 3.3, \( S_t.k[i] \leq \lceil \lg n \rceil \) or
$S_t.k[j] \leq \lceil \lg n \rceil$. Since $p_a$ in Critical region in state $S$ implies $S.k[a] = \lceil \lg n \rceil + 1$, at least one of $\{p_i, p_j\}$ is not in the Critical region at state $S_i$. $\blacksquare$
Chapter 4

Conclusion

We have shown that the Peterson-Fischer 2-process and n-process mutual exclusion algorithms satisfy mutual exclusion. This alone is a significant result, but also interesting is the strategy of the proof. The n-process algorithm, conceptually, is a tournament of 2-process competitions. One can see this from looking at the 2-process code and the n-process code side by side. However, in this proof we have successfully formalized this construction.

First, we extended the 2-process algorithm to allow a greater set of starting states. Namely, we allowed $p_0$ of the 2-process system to start anywhere in its code, with some restriction on the starting value of $q[0]$. This extension was necessary for the mapping (that appeared in the n-process proof) between the n-process states and the 2-process states, and it was done with hindsight.

Next, we formally defined the state of the 2-process system. To prove that the 2-process algorithm satisfies mutual exclusion, we used an invariant-assertional technique. We stated a series of properties that hold for all reachable states, culminating in the final invariant of mutual exclusion.

After the 2-process algorithm was shown to satisfy mutual exclusion, we then began the n-process proof by defining the state of the n-process system. This state definition, along with the 2-process state definition, was a keystone of the proof because our strategy was to develop a mapping between the states of the two systems.

Then, during the proof of the n-process algorithm, we used an inductive argument. This allowed us to focus on two processes, $p_i$ and $p_j$, during a segment of the execution, $[S_t, S_i]$. 

33
Conceptually, this segment corresponded to one 2-process competition, between \( p_i \) and \( p_j \), in the \( n \)-process tournament.

Finally, after all of the preceding groundwork, we developed the mapping between states of the \( n \)-process system in the interval \([S_{t_j}, S_t]\) and states of the 2-process system. Using this mapping, we were able to reduce that section of the \( n \)-process execution to an analogous execution of a corresponding 2-process system. In this way, we were able to use proven statements about the 2-process system to show properties about the \( n \)-process system. The 2-process system was used not only as a building block of the \( n \)-process algorithm, but also as a building block of the \( n \)-process mutual exclusion proof that we have presented here. In this way, the 2-process mutual exclusion proof acts as a “subroutine” of the \( n \)-process mutual exclusion proof, much as the 2-process algorithm is used as a subroutine of the \( n \)-process algorithm.

The significance of this technique lies in the fact that correctness proofs of algorithms are often difficult to structure in a modular style. Here, we carefully proved, using state invariants, one simple algorithm, and we then showed how that proof can be used as a module in a proof of a complex algorithm with the addition of a state mapping.

Future work in this area would begin with liveness proofs for the 2-process and \( n \)-process algorithms. Perhaps they, too, could make use of a similar modular construction.
Bibliography