Verifying Mutual Exclusion and Liveness Properties with TLA

Ioana–Cleopatra Pau
Igor Rents
Wolfgang Schreiner

January 28, 2000

Abstract

In this paper we prove the correctness of Peterson's mutual exclusion algorithm using the Temporal Logic of Actions (TLA). We present a precise analysis of the algorithm for two processes: the formal specification of the algorithm, of the mutual exclusion property and of the liveness property and the proof that the algorithm satisfies these two properties.

1 Introduction

This paper is the result of our work in the "Parallel and Distributed Software and Algorithms" seminar held at RISC-Linz during the winter semester 1998/1999. We prove the correctness of a distributed algorithm using the Temporal Logic of Actions [1].

We present in this paper a precise analysis of Peterson's mutual exclusion algorithm for two processes [2]: the formal specification of the algorithm, of the mutual exclusion property and of the liveness property and the proof that the algorithm satisfies these two properties.

When one develops algorithms which are intended to be executed in a concurrent way, it is very important to be sure that the algorithms are correct whatever the interleaving order of the concurrent processes is. Even for a simple distributed algorithm, this analysis is difficult to carry out. For more complicated algorithm this task might be impossible without a precise way of reasoning. That is why our goal in this paper is not only to prove some properties of the algorithm, but also to formalize the proof rigorously.

The mutual exclusion property of Peterson's algorithm was thoroughly considered in [3] in the frame of the linear time temporal logic. Though we prove the property in the frame of TLA and the details of the proof are different, the two approachs are similar. As in any proof of a safety property of a program it is necessary to find appropriate inductive invariant assertions during the proof.

We use in this paper a formal language for proofs, completed by graphical representation and comments in natural language. In Section 2 we present the problem and the algorithm in an informal way and also the procedural description of the algorithm. Section 3 is dedicated to the presentation of the formal language that we use and to the formalization of the algorithm. In Sections 4 and 5 we give the proofs for the mutual exclusion and liveness properties, respectively.

2 Peterson's Mutual Exclusion Algorithm

We consider the following problem: given two processes and a shared resource, which must not be accessed by these two processes simultaneously (for example a printer), we have to write a program that allows each process to access the resource.

The mutual exclusion property ensures that these two processes cannot access the shared resource at the same time. The liveness property is ensured by the lockout freedom, which means that if both processes always return the resource, then any process requesting the resource will eventually get it.

Algorithm 2.1 Peterson's mutual exclusion algorithm, $(i \in \{0,1\})$

loop forever

set_flag:	$flag_i := 1$
set_turn:	$turn := i$
$wait$:	waitfor $flag_{\bar{i}} = 0$ or turn $\neq i$
$crit$:	(* critical region *)
$reset\_flag$:	$flag_i := 0$
rem:	(* remainder region *)

In the algorithm critical region and remainder region mean some commands executed when the process works with the shared resource and between the release of this resource and another request of it, respectively.

The idea of the algorithm is very simple (see Algorithm 2.1). In this algorithm we use the following shared variables:

$$turn \in \{0,1\}$$
$$flag_i \in \{0,1\}, \text{ where } i \in \{0,1\}$$

These variables are readable by both processes, $turn$ is also writable by both processes, but $flag_i$ is writable only by process i. Every process i has a flag associated $flag_i$ (we denote by $\bar{i}$ the number of the other process). The value of the flag is initially 0. When a process requests the resource it sets its flag to 1 and it remains with this value until the process returns the resource. In this way a process can know whether the other process does not need the resource. For the case in which both processes need the resource at the same time, the variable $turn$ is used to store the number of the last process that requested the resource. A process i can get the resource only either the other one does not need it, i.e., $flag_{\bar{i}} = 0$, or the other process requested the resource after process i, i.e., $turn \neq i$.

3 Formalization of the Algorithm in TLA

Here we present a formalization of the algorithm (see figure 3.1) in a slightly modified Temporal Logic of Actions developed by Lamport [1]. Our aim is to construct some formula Φ of the logic describing the algorithm.

A state in our system is defined by the values of the variables. To be able to reason about parts of the code we introduce two additional variables pc_0 and pc_1. Intuitively, we can think about them as the "program counters", referencing the next atomic statement to be executed. As in Lamport's TLA we have two sets of variables: unprimed ones, i.e.,

$$Vars = \{flag_0, flag_1, turn, pc_0, pc_1\},$$

that reflects the current state of the system, and primed ones, i.e.,

$$Vars' = \{flag_0', flag_1', turn', pc_0', pc_1'\},$$

that reflects the next state of the system (after execution of the statement). To shorten our writings we introduce the predicate $\mathcal{U}(V)$, which says that all the variables not in the set V remain unchanged.

Figure 3.1 Formalization of the algorithm in TLA

$$Vars = \{flag_0, flag_1, turn, pc_0, pc_1\},\ Vars' = \{flag_0', flag_1', turn', pc_0', pc_1'\}$$

$$\mathcal{U}(V) \equiv \bigwedge_{v \in Vars - V} v' = v$$

$$\Phi \equiv Init \wedge \Box Actions \wedge WF$$

$$\begin{aligned}
&Init \equiv flag_0 = 0 \wedge flag_1 = 0 \wedge (turn = 0 \vee turn = 1) \wedge \\
&\qquad \wedge pc_0 = rem \wedge pc_1 = rem \\
&Actions \equiv Actions_0 \vee Actions_1
\end{aligned}$$

$$\begin{aligned}
&Actions_i \equiv Set_flag_i \vee Set_turn_i \vee Wait_i \vee Crit_i \vee Reset_flag_i \vee Rem_i \\
&Set_flag_i \equiv pc_i = set_flag \wedge flag_i' = 1 \wedge pc_i' = set_turn \wedge \mathcal{U}(\{flag_i, pc_i\}) \\
&Set_turn_i \equiv pc_i = set_turn \wedge turn' = i \wedge pc_i' = wait \wedge \mathcal{U}(\{turn, pc_i\}) \\
&Wait_i \equiv pc_i = wait \wedge (flag_{\bar{i}} = 0 \vee turn \neq i) \wedge pc_i' = crit \wedge \mathcal{U}(\{pc_i\}) \\
&Crit_i \equiv pc_i = crit \wedge pc_i' = reset_flag \wedge \mathcal{U}(\{pc_i\}) \\
&Reset_flag_i \equiv pc_i = reset_flag \wedge flag_i' = 0 \wedge pc_i' = rem \wedge \mathcal{U}(\{flag_i, pc_i\}) \\
&Rem_i \equiv pc_i = rem \wedge pc_i' = set_flag \wedge \mathcal{U}(\{pc_i\})
\end{aligned}$$

$$\begin{aligned}
&WF \equiv WF_0 \wedge WF_1 \\
&WF_i \equiv WF_i(Set_flag_i) \wedge WF_i(Set_turn_i) \wedge WF_i(Wait_i) \wedge WF_i(Crit_i) \\
&\qquad \wedge WF_i(Reset_flag_i) \wedge WF_i(Rem_i) \\
&WF_i(A) \equiv \Box\Diamond A \vee \Box\Diamond\neg Enabled(A)
\end{aligned}$$

$$\begin{aligned}
&Enabled(Set_flag_i) \equiv pc_i = set_flag \\
&Enabled(Set_turn_i) \equiv pc_i = set_turn \\
&Enabled(Wait_i) \equiv pc_i = wait \wedge (flag_{\bar{i}} = 0 \vee turn \neq i) \\
&Enabled(Crit_i) \equiv pc_i = crit \\
&Enabled(Reset_flag_i) \equiv pc_i = reset_flag \\
&Enabled(Rem_i) \equiv pc_i = rem
\end{aligned}$$

The initial state condition $Init$ describes the values of all variables in the first state of the system. In our logic we represent actions by predicates relating the values of the system in any two subsequent states. An action is enabled in a state S_1 if there is a state S_2 such that the action describes the transition from S_1 to S_2.

The temporal formula $\Box A$ ("always A") is said to be true in some state iff A is true in any subsequent state. We consider the formula $\Diamond A$ as a shortcut for $\neg\Box\neg A$.

We will use the sign $\leadsto$ for denoting the operator *leadsto*: for F and G temporal formulas, F leads to G iff, at any time when F is true, G is true then or at some later time:

$$F \leadsto G :\Leftrightarrow \Box(F \to \Diamond G).$$

This operator is transitive (see [1]).

4 The Mutual Exclusion Property

In this section we want to prove that the algorithm satisfies mutual exclusion property, i.e., these two processes cannot be in their critical regions simultaneously. In our formal language it can be expressed by the formula: $\Box\neg(pc_0 = crit \wedge pc_1 = crit)$. To show that this algorithm satisfies mutual exclusion property we have to prove: $\Phi \to \Box\neg(pc_0 = crit \wedge pc_1 = crit)$. But first, we show that the variables can obtain only meanings from their domains.

Proposition 1. *For all $j \in \{0, 1\}$:*

a) $\Phi \to \Box(pc_j = set\_flag \vee pc_j = set\_turn \vee pc_j = wait \vee pc_j = crit$
$\qquad \vee pc_j = reset\_flag \vee pc_j = rem)$

b) $\Phi \to \Box(flag_j = 0 \vee flag_j = 1)$

c) $\Phi \to \Box(turn = 0 \vee turn = 1)$

Proof. The proof is simple. $\Box$

We define two assertions and prove that they are invariant assertions of the algorithm.

Definition 1. *For all $j \in \{0, 1\}$ let us define*
$I_1^j \equiv (flag_j = 0) \leftrightarrow (pc_j = rem \vee pc_j = set\_flag)$
$I_2^j \equiv (pc_j = crit \vee pc_j = reset\_flag) \wedge (pc_{\overline{j}} = wait \vee pc_{\overline{j}} = crit$
$\qquad \vee pc_{\overline{j}} = reset\_flag) \to turn = \overline{j}$.

Proposition 2. *For $j \in \{0, 1\}$:* $\Phi \to \Box I_1^j$.

Proposition 3. *For $j \in \{0, 1\}$:* $\Phi \to \Box I_2^j$.

The proofs of both propositions are given at the end of the section.

Now we are ready to show that the algorithm satisfies mutual exclusion property.

Theorem 1 (Mutual Exclusion). $\Phi \to \Box\neg(pc_0 = crit \wedge pc_1 = crit)$

Proof. From proposition 3 we have

$$\Phi \to \Box I_2^0 \tag{1}$$

and

$$\Phi \rightarrow \Box I_2^1 \tag{2}$$

Assume

$$\Phi \tag{3}$$

Prove

$$\Box\neg(pc_0 = crit \wedge pc_1 = crit) \tag{4}$$

From 1 and 3 we obtain $\Box I_2^0$, i.e.,

$$\begin{aligned} &\Box((pc_0 = crit \vee pc_0 = reset\_flag) \\ \wedge&(pc_1 = wait \vee pc_1 = crit \vee pc_1 = reset\_flag) \rightarrow turn = 1) \end{aligned} \tag{5}$$

From 2 and 3 we obtain $\Box I_2^1$, i.e.,

$$\begin{aligned} &\Box((pc_1 = crit \vee pc_1 = reset\_flag) \\ \wedge&(pc_0 = wait \vee pc_0 = crit \vee pc_0 = reset\_flag) \rightarrow turn = 0) \end{aligned} \tag{6}$$

We prove the formula 4 in two steps:

(a) prove $\neg(pc_0 = crit \wedge pc_1 = crit)$

(b) prove $\neg({pc_0}' = crit \wedge {pc_1}' = crit)$

Show a: Suppose

$$pc_0 = crit \tag{7}$$

and

$$pc_1 = crit \tag{8}$$

From 5,7 and 8 we obtain

$$turn = 1 \tag{9}$$

From 6,7 and 8 we obtain

$$turn = 0 \tag{10}$$

From 9 and 10 we obtain a contradiction.

Show b: Suppose

$${pc_0}' = crit \tag{11}$$

and

$${pc_1}' = crit \tag{12}$$

From 5,11 and 12 we obtain

$$turn' = 1 \tag{13}$$

From 6,11 and 12 we obtain

$$turn' = 0 \tag{14}$$

From 13 and 14 we obtain a contradiction.

$\square$

Below proofs of the propositions 2 and 3 are given.

Proof (Proposition 2). Take j arbitrary. We prove the formula in two steps:

(a) prove $Init \rightarrow I_1^j$

(b) prove $I_1^j \wedge Actions \rightarrow {I_1^j}'$

Show a: Assume that Init holds, i.e.,

$$flag_j = 0 \wedge flag_{\overline{j}} = 0 \wedge (turn = 0 \vee turn = 1) \\ \wedge pc_j = rem \wedge pc_{\overline{j}} = rem \tag{1}$$

Prove I_1^j, i.e.,

$$(flag_j = 0) \leftrightarrow (pc_j = rem \vee pc_j = set\_flag) \tag{2}$$

From 1 we obtain:

$$flag_j = 0 \tag{3}$$

and

$$pc_j = rem \tag{4}$$

From 3 and 4 we obtain 2.

Show b: Assume that I_1^j holds, i.e.,

$$(flag_j = 0) \leftrightarrow (pc_j = rem \vee pc_j = set\_flag) \tag{5}$$

Assume that $Actions$ holds, i.e.,

$$Set\_flag_j \vee Set\_turn_j \vee Wait_j \vee Crit_j \vee \\ Reset\_flag_j \vee Rem_j \vee Set\_flag_{\overline{j}} \vee \vee Set\_turn_{\overline{j}} \vee \\ Wait_{\overline{j}} \vee Crit_{\overline{j}} \vee Reset\_flag_{\overline{j}} \vee Rem_{\overline{j}} \tag{6}$$

We want to prove ${I_1^j}'$, i.e.,

$$({flag_j}' = 0) \leftrightarrow ({pc_j}' = rem \vee {pc_j}' = set\_flag) \tag{7}$$

To prove 7 we have to consider twelve cases due to the disjunctive formula 6. However, actions of process $\overline{j}$ do not change $flag_j$ and pc_j. Thus using 5 for these six cases, we obtain that 7 is obviously true. So we have to consider only the other six actions of process j.

Set_flag_j**:** Assume that Set_flag_j holds, i.e.,

$$pc_j = set\_flag \wedge flag_j' = 1 \wedge pc_j' = set\_turn \wedge \\ pc_{\overline{j}}' = pc\overline{j} \wedge flag_{\overline{j}}' = flag_{\overline{j}} \wedge turn' = turn \tag{8}$$

From 8 we obtain:

$$flag_j' = 1 \tag{9}$$

and

$$pc_j' = set\_turn \tag{10}$$

From 9 and 10 we obtain 7.

Set_turn_j: Assume that Set_turn_j holds, i.e.,

$$pc_j = set\_turn \wedge turn' = j \wedge pc'_j = wait \wedge \\ flag'_j = flag_j \wedge flag'_{\overline{j}} = flag_{\overline{j}} \wedge pc'_{\overline{j}} = pc_{\overline{j}} \tag{11}$$

From 11 we obtain:

$$pc'_j = wait \tag{12}$$

and

$$pc_j = set\_turn \tag{13}$$

From proposition 1b, 13 and 5 we obtain:

$$flag_j = 1 \tag{14}$$

From 11 and 14 we obtain:

$$flag'_j = 1 \tag{15}$$

From 15 and 12 we obtain 7.

$Wait_j$: Assume that $Wait_j$ holds, i.e.,

$$pc_j = wait \wedge (flag_{\overline{j}} = 0 \vee turn \neq 0) \wedge pc'_j = crit \wedge \\ flag'_j = flag_j \wedge flag'_{\overline{j}} = flag_{\overline{j}} \wedge \\ turn' = turn \wedge pc'_{\overline{j}} = pc_{\overline{j}} \tag{16}$$

From 16 we obtain:

$$pc'_j = crit \tag{17}$$

and

$$pc_j = wait \tag{18}$$

From proposition 1b, 5 and 18 we obtain:

$$flag_j = 1 \tag{19}$$

From 19 and 16 we obtain:

$$flag'_j = 1 \tag{20}$$

From 17 and 20 we obtain 7.

$Crit_j$: Assume that $Crit_j$ holds, i.e.,

$$pc_j = crit \wedge pc'_j = reset\_flag \wedge flag'_j = flag_j \wedge \\ flag'_{\overline{j}} = flag_{\overline{j}} \wedge turn' = turn \wedge pc'_{\overline{j}} = pc_{\overline{j}} \tag{21}$$

From 21 we obtain:

$$pc_j = crit \tag{22}$$

and

$$pc'_j = reset\_flag \tag{23}$$

From proposition 1b, 5 and 22 we obtain:

$$flag_j = 1 \tag{24}$$

From 21 and 24 we obtain:

$$flag'_j = 1 \tag{25}$$

From 23 and 25 we obtain 7.

$Reset\_flag_j$**:** Assume that $Reset\_flag_j$ holds, i.e.,

$$pc_j = reset\_flag \wedge flag'_j = 0 \wedge pc'_j = rem \wedge \\ flag'_{\overline{j}} = flag_{\overline{j}} \wedge pc'_{\overline{j}} = pc_{\overline{j}} \wedge turn' = turn \tag{26}$$

From 26 we obtain:

$$flag'_j = 0 \tag{27}$$

and

$$pc'_j = rem \tag{28}$$

From 27 and 28 we obtain 7.

Rem_j**:** Assume that Rem_j holds, i.e.,

$$pc_j = rem \wedge pc'_j = set\_flag \wedge pc'_{\overline{j}} = pc_{\overline{j}} \wedge \\ flag'_j = flag_j \wedge flag'_{\overline{j}} = flag_{\overline{j}} \wedge turn' = turn \tag{29}$$

From 29 we obtain:

$$pc_j = rem \tag{30}$$

and

$$pc'_j = set\_flag \tag{31}$$

From 5 and 30 we obtain:

$$flag_j = 0 \tag{32}$$

From 29 and 32 we obtain:

$$flag'_j = 0 \tag{33}$$

From 31 and 33 we obtain 7.

□

Proof (Proposition 3). Take arbitrary j. To prove $\Phi \rightarrow \Box I_2^j$ we have to prove that

(a) $Init \rightarrow I_2^j$ and

(b) $I_1^j \wedge I_1^{\overline{j}} \wedge I_2^{\overline{j}} \wedge I_2^j \wedge Actions \rightarrow {I_2^j}'$

Show a: Assume Init holds, i.e.,

$$flag_0 = 0 \wedge flag_1 = 0 \wedge (turn = 0 \vee turn = 1) \\ \wedge pc_0 = rem \wedge pc_1 = rem \tag{1}$$

We have to prove I_2^j, i.e.,

$$(pc_j = crit \vee pc_j = reset\_flag) \wedge (pc_{\overline{j}} = wait \vee \\ pc_{\overline{j}} = crit \vee pc_{\overline{j}} = reset\_flag) \rightarrow turn = \overline{j} \tag{2}$$

From 1 we obtain:

$$pc_j = rem \tag{3}$$

From 3 we obtain 2.

Show b: Assume $I_1^j \wedge I_1^{\overline{j}} \wedge I_2^{\overline{j}} \wedge I_2^j \wedge Actions$ holds, i.e.,

$$(flag_j = 0) \leftrightarrow (pc_j = rem \vee pc_j = set\_flag) \tag{4}$$

$$(flag_1 = j) \leftrightarrow (pc_{\overline{j}} = rem \vee pc_{\overline{j}} = set\_flag) \tag{5}$$

$$\begin{aligned}(pc_j = crit \vee pc_j = reset\_flag) \wedge (pc_{\overline{j}} = wait \vee pc_{\overline{j}} = crit \\ \vee pc_{\overline{j}} = reset\_flag) \rightarrow turn = \overline{j}\end{aligned} \tag{6}$$

$$\begin{aligned}(pc_{\overline{j}} = crit \vee pc_{\overline{j}} = reset\_flag) \wedge (pc_j = wait \vee pc_j = crit \vee \\ \vee pc_j = reset\_flag) \rightarrow turn = j\end{aligned} \tag{7}$$

$$\begin{aligned}Set\_flag_0 \vee Set\_turn_0 \vee Wait_0 \vee Crit_0 \vee Reset\_flag_0 \\ \vee Rem_0 \vee Set\_flag_1 \vee Set\_turn_1 \vee \\ Wait_1 \vee Crit_1 \vee Reset\_flag_1 \vee Rem_1\end{aligned} \tag{8}$$

We have to prove ${I_2^j}'$, i.e.,

$$\begin{aligned}(pc'_j = crit \vee pc'_j = reset\_flag) \wedge (pc'_{\overline{j}} = wait \vee pc'_{\overline{j}} = crit \vee \\ \vee pc'_{\overline{j}} = reset\_flag) \rightarrow turn' = \overline{j}\end{aligned} \tag{9}$$

To prove 9 we have to consider twelve cases due to the disjunctive formula 8. However, it is enough to consider only five cases instead of twelve: $Wait_j, Crit_j, Set\_turn_{\overline{j}}, Wait_{\overline{j}}, Crit_{\overline{j}}$. For other cases we have either $pc'_j \neq crit \wedge pc'_j \neq reset\_flag$ or $pc'_{\overline{j}} \neq wait \wedge pc'_{\overline{j}} \neq crit \wedge \wedge pc'_{\overline{j}} \neq reset\_flag$ and it makes 9 to hold.

Case $Wait_j$: Assume $Wait_j$ holds, i.e.,

$$\begin{aligned}pc_j = wait \wedge (flag_{\overline{j}} = 0 \vee turn \neq j) \wedge \\ pc'_j = crit \wedge pc'_{\overline{j}} = pc_{\overline{j}} \wedge flag'_{\overline{j}} = flag_{\overline{j}} \wedge \\ flag'_j = flag_j \wedge turn' = turn\end{aligned} \tag{10}$$

From 10 we obtain:

$$flag_{\overline{j}} = 0 \vee turn \neq j \tag{11}$$

Case $flag_{\overline{j}} = 0$:

$$flag_{\overline{j}} = 0 \tag{12}$$

From 5 and 12 we obtain:

$$(pc_{\overline{j}} = rem \vee pc_{\overline{j}} = set\_flag) \tag{13}$$

From 13 we obtain 9

Case $turn \neq j$**:**

$$turn \neq j \tag{14}$$

From proposition 1c and 14 we obtain:

$$turn = \overline{j} \tag{15}$$

From 10 and 15 we obtain:

$$turn' = \overline{j} \tag{16}$$

From 16 we obtain 9.

Case $Crit_j$**:** Assume $Crit_j$ holds, i.e.,

$$\begin{aligned} pc_j = crit \wedge pc'_j = reset\_flag \wedge pc'_{\overline{j}} = pc_{\overline{j}} \\ \wedge flag'_j = flag_j \wedge flag'_{\overline{j}} = flag_{\overline{j}} \wedge turn' = turn \end{aligned} \tag{17}$$

From 17 we obtain:

$$pc_j = crit \tag{18}$$

Case $pc_{\overline{j}} = wait \vee pc_{\overline{j}} = crit \vee pc_{\overline{j}} = reset\_flag$, i.e.,

$$pc_{\overline{j}} = wait \vee pc_{\overline{j}} = crit \vee pc_{\overline{j}} = reset\_flag \tag{19}$$

From 6, 18 and 19 we obtain:

$$turn = \overline{j} \tag{20}$$

From 17 and 20 we obtain:

$$turn' = \overline{j} \tag{21}$$

From 21 we obtain 9

Case $pc_{\overline{j}} \neq wait \wedge pc_{\overline{j}} \neq crit \wedge pc_{\overline{j}} \neq reset\_flag$, i.e.,

$$pc_{\overline{j}} \neq wait \wedge pc_{\overline{j}} \neq crit \wedge pc_{\overline{j}} \neq reset\_flag \tag{22}$$

From 17 and 22 we obtain:

$$pc'_{\overline{j}} \neq wait \wedge pc'_{\overline{j}} \neq crit \wedge pc'_{\overline{j}} \neq reset\_flag \tag{23}$$

From 23 we obtain 9.

Case $Set\_turn_{\overline{j}}$ Assume $Set\_turn_{\overline{j}}$ holds, i.e.,

$$\begin{aligned} pc_{\overline{j}} = set\_turn \wedge turn' = \overline{j} \wedge pc'_{\overline{j}} = wait \wedge pc'_j = pc_j \\ \wedge flag'_j = flag_j \wedge flag'_{\overline{j}} = flag_{\overline{j}} \end{aligned} \tag{24}$$

From 24 we have:

$$turn' = \overline{j} \tag{25}$$

From 25 we obtain 9.

Case $Wait_{\overline{j}}$**:** Assume $Wait_{\overline{j}}$ holds, i.e.,

$$\begin{aligned} pc_{\overline{j}} = wait \wedge (flag_j = 0 \vee turn = j) \wedge \\ pc'_{\overline{j}} = crit \wedge pc'_j = pc_j \wedge flag'_j = flag_j \\ \wedge flag'_{\overline{j}} = flag_{\overline{j}} \wedge turn' = turn \end{aligned} \tag{26}$$

Case $pc_j \neq crit \wedge pc_j \neq reset\_flag,$ i.e.,

$$pc_j \neq crit \wedge pc_j \neq reset\_flag \tag{27}$$

From 26 and 27 we obtain:

$$pc'_j \neq crit \wedge pc'_j \neq reset\_flag \tag{28}$$

From 28 we obtain 9.

Case $pc_j = crit \vee pc_j = reset\_flag,$ i.e.,

$$pc_j = crit \vee pc_j = reset\_flag \tag{29}$$

From 26 we obtain:

$$pc_{\overline{j}} = wait \tag{30}$$

From 6, 29 and 30 we obtain:

$$turn = \overline{j} \tag{31}$$

From 26 and 31 we obtain:

$$turn' = \overline{j} \tag{32}$$

From 32 we obtain 9.

Case $Crit_{\overline{j}}$**:** Assume $Crit_{\overline{j}}$ holds, i.e.,

$$\begin{aligned} pc_{\overline{j}} = crit \wedge pc'_{\overline{j}} = reset\_flag \wedge flag'_j = flag_j \\ \wedge flag'_{\overline{j}} = flag_{\overline{j}} \wedge pc'_j = pc_j \wedge turn' = turn \end{aligned} \tag{33}$$

Case $pc_j \neq crit \wedge pc_j \neq reset\_flag$, i.e.,

$$pc_j \neq crit \wedge pc_j \neq reset\_flag \tag{34}$$

From 33 and 34 we obtain:

$$pc'_j \neq crit \wedge pc'_j \neq reset\_flag \tag{35}$$

From 35 we obtain 9

Case $pc_j = crit \vee pc_j = reset\_flag$, i.e.,

$$pc_j = crit \vee pc_j = reset\_flag \tag{36}$$

From 33 we obtain:

$$pc_{\overline{j}} = crit \tag{37}$$

From 6, 36 and 37 we obtain:

$$turn = \overline{j} \tag{38}$$

From 33 and 38 we obtain:

$$turn' = \overline{j} \tag{39}$$

From 39 we obtain 9.

□

5 The Lockout Freedom Property

In the previous section we have proven the mutual exclusion property: if a process is in the critical region, the other process cannot be at the same time in the critical region. However, this does not guarantee that a process which is waiting to access the critical region will eventually access it. It is possible that at a certain moment none of the system variables changes anymore, i.e. only stuttering steps are performed.

To avoid such a behavior we have introduced weak fairness conditions in the formal specification of the algorithm (see figure 5.1). This will ensure that, eventually, an action will be executed, or it will not be enabled.

For a clear understanding of the system's behavior we will present it also in a graphical way using *verification diagrams* (see [3]).

Definition 1. *A verification diagram is a directed labeled graph constructed in the following way:*

- *A node represents one or more states of the system, characterized by an assertion. We label the node with the corresponding assertion.*
- *An edge represents an action between two states and is labeled by the name of the action.*

We also use some conventions introduced in [3], in order to improve the readability of a diagram. We will use compound nodes containing internal nodes. The label of a compound node is considered as though it were a conjunct added to each label of a component node. An edge departing from a compound node is considered as though it departed from each component node and, in a similar way, an edge arriving at a compound node is considered as though it arrived at each component node. We do not represent the edges that connect a node to itself.

From the diagram it can be seen that every action executed by process i, except Set_turn_i, enables the next action. Also, none of the actions except $Wait_i$ can be disabled by the execution of another enabled action. This can be easily proven formally.

The only non obvious behavior of the process i occurs when it is waiting to enter in the critical region. In this case $Wait_i$ is not always enabled; even when it is enabled, it might be disabled before being executed.

We will prove that, in our program, a state in which the process i is waiting to access the critical region, leads to a state in which the process i is in the critical region. A general strategy for proving that, $P \leadsto Q$, for two states P, Q, in a program with weak fairness property, is to show that, once in the state P, the program keeps the same state for a finite number of steps, until eventually an enabled action changes the state to Q.

Formally, for proving $\Box Actions \wedge WF(A) \Rightarrow (P \leadsto Q)$, where A is the "helpful" action, it is sufficient to prove (see [1]:

(a) $Actions \wedge P \Rightarrow P' \vee Q'$;

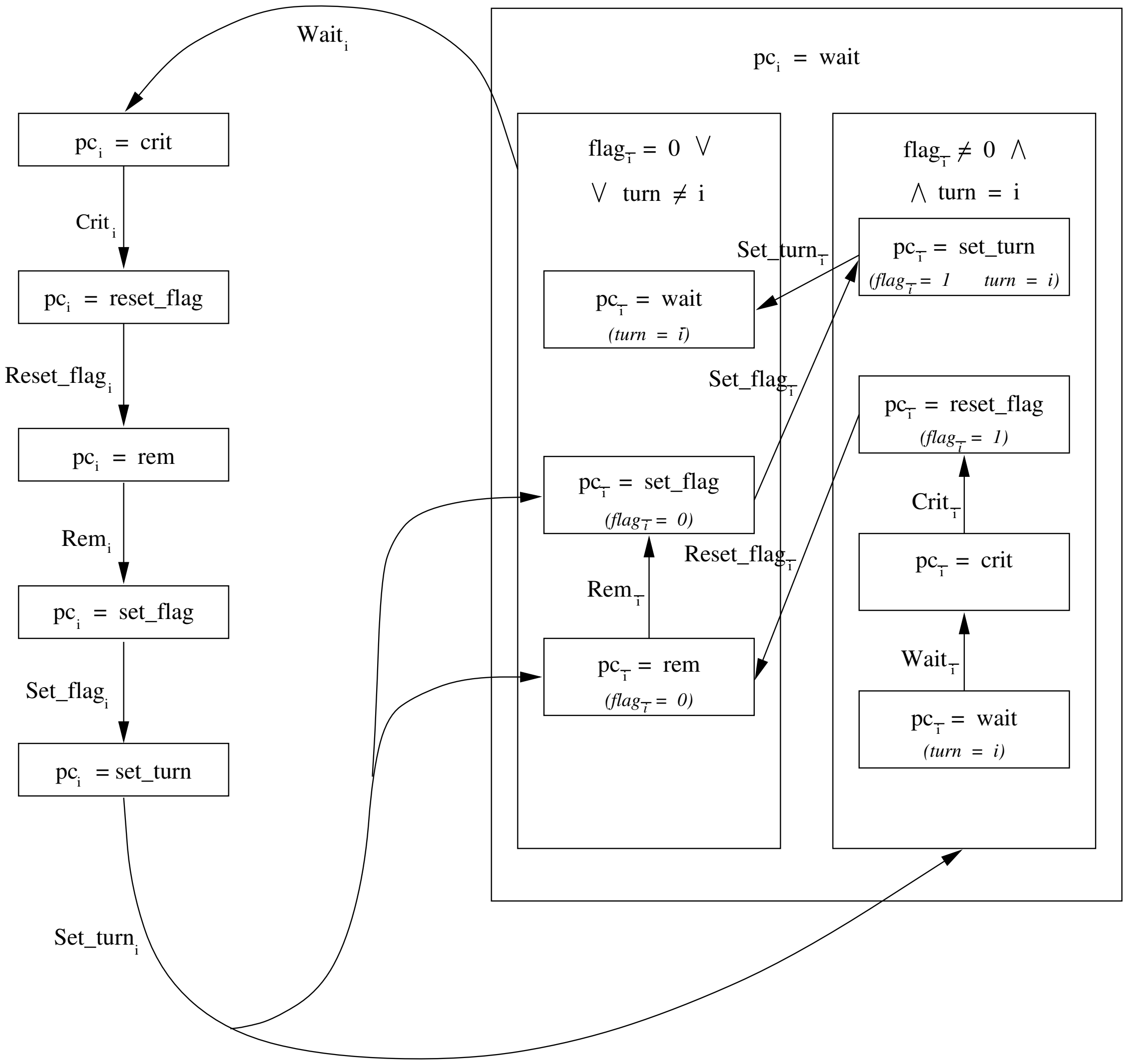
(b) $(Actions \wedge A) \wedge P \Rightarrow Q'$;

(c) $P \Rightarrow Enabled(A)$.

Proposition 4. $\Phi_{WF} \Rightarrow (pc_i = wait) \leadsto (pc_i = crit)$

Proof. In order to prove $\Phi_{WF} \Rightarrow (pc_i = wait) \leadsto (pc_i = crit)$, we prove:

Figure 5.1 The verification diagram representing the progress of the algorithm



1. $\Phi_{WF} \Rightarrow (pc_i = wait \wedge Enabled(Wait_i)) \leadsto (pc_i = crit)$

and

2. $\Phi_{WF} \Rightarrow (pc_i = wait \wedge \neg Enabled(Wait_i)) \leadsto (pc_i = crit)$

1. Let $P_1 \equiv pc_i = wait \wedge (flag_{\overline{\imath}} = 0 \vee turn \neq i)$ and $Q \equiv pc_i = crit$.

   From the diagram one can easily see that in a state where P_1 holds, on the execution of any action other than $Set\_flag_{\overline{\imath}}$, the system remains in such a state or reaches directly a state where Q holds. For this reason we will prove separately:

   i. $\Phi_{WF} \Rightarrow (pc_i = wait \wedge Enabled(Wait_i) \wedge \neg Set\_flag_{\overline{\imath}}) \leadsto (pc_i = crit)$ and

   ii. $\Phi_{WF} \Rightarrow (pc_i = wait \wedge Enabled(Wait_i) \wedge Set\_flag_{\overline{\imath}}) \leadsto (pc_i = crit)$

   i. We are going to prove (a-c) with $A \equiv Wait_i$ and $P \equiv P_1 \wedge \neg Set\_flag_{\overline{\imath}}$. Because

   $$Wait_i \wedge P \Rightarrow Q',$$

   by the definition of $Wait_i$, it is easy to prove that (b) holds. It is obvious that (c) holds, too.

   We will prove in detail that (a) also holds. Assume $Actions \wedge P$. We will prove $P' \vee Q'$.

   Case $Actions_i$. None of the actions of the process i except $Wait_i$ is enabled, thus the only possible action is $Wait_i$. By definition we have:

   $$Wait_i \wedge P \Rightarrow Q',$$

   from which it follows immediately (a).

   Case $Actions_{\overline{\imath}}$. Take $Action_{\overline{\imath}} \in \{Wait_{\overline{\imath}}, Crit_{\overline{\imath}}, Rem_{\overline{\imath}}\}$. In this case $Action_{\overline{\imath}} \wedge P \Rightarrow P'$, because of $\mathcal{U}(\{pc_{\overline{\imath}}\})$. The action $Set\_flag_{\overline{\imath}}$ cannot be executed because it contradicts P, thus we consider only $Action_{\overline{\imath}} \in \{Set\_turn_{\overline{\imath}}, Reset\_flag_{\overline{\imath}}\}$.

   When we consider $Set\_turn_{\overline{\imath}}$, we have by definition:

   $$Set\_turn_{\overline{\imath}} \wedge P \Rightarrow turn' = \overline{\imath} \wedge pc_i' = wait,$$

   thus $Set\_turn_{\overline{\imath}} \wedge P \Rightarrow P'$.

   In the case $Reset\_flag_{\overline{\imath}}$, by definition:

   $$Reset\_flag_{\overline{\imath}} \wedge P \Rightarrow flag_{\overline{\imath}}' = 0 \wedge pc_i' = wait,$$

   i.e. $Reset\_flag_{\overline{\imath}} \wedge P \Rightarrow P'$.

   ii. Let $P \equiv P_1 \wedge Set\_flag_{\overline{\imath}}$. Here we cannot prove the *leads-to* relation directly. We therefore assume Φ_{WF} and prove the following chain of *leads-to* relations:

   $$(Set\_flag_{\overline{\imath}} \wedge pc_i = wait) \leadsto (pc_{\overline{\imath}} = set\_turn \wedge pc_i = wait) \leadsto$$

   $$\leadsto (pc_{\overline{\imath}} = wait \wedge pc_i = wait \wedge turn = \overline{\imath}) \leadsto (pc_i = crit),$$

   with the "helpful" action $Set\_flag_{\overline{\imath}}$, $Set\_turn_{\overline{\imath}}$, $Wait_i$ respectively. By transitivity of $\leadsto$, then $(Set\_flag_{\overline{\imath}} \wedge pc_i = wait) \leadsto (pc_i = crit)$ holds.

- $(Set\_flag_{\bar{\imath}} \wedge pc_i = wait) \leadsto (pc_{\bar{\imath}} = set\_turn \wedge pc_i = wait)$. By the definition of $Set\_flag_{\bar{\imath}}$ and of $Enabled(Set\_flag_{\bar{\imath}})$, (a), (b), (c) hold easily. We observe that after the execution of $Set\_flag_{\bar{\imath}}$, $Wait_i$ is not enabled anymore, because $flag'_{\bar{\imath}} = 1 \wedge turn' = i$.
- $(pc_{\bar{\imath}} = set\_turn \wedge pc_i = wait) \leadsto (pc_{\bar{\imath}} = wait \wedge pc_i = wait \wedge turn = \bar{\imath})$. This is also easy to prove by the definition of $Set\_turn_{\bar{\imath}}$ and of $Enabled(Set\_turn_{\bar{\imath}})$. No other action is enabled in this case, and after the execution of $Set\_turn_{\bar{\imath}}$, the action $Wait_i$ will become enabled (because $turn' = \bar{\imath}$).
- $(pc_{\bar{\imath}} = wait \wedge pc_i = wait \wedge \bar{\imath}) \leadsto (pc_i = crit)$. Because of the fact that the only action enabled in this case is $Wait_i$, the relation follows from the case (i) proven above.

2. We assume Φ_{WF} and will prove $R \leadsto Q$, where $R \equiv flag_{\bar{\imath}} \neq 0 \wedge turn = i \wedge pc_i = wait$. We cannot prove this directly, but will use the transitivity. We consider the possible of $pc_{\bar{\imath}}$.

   Case $pc_{\bar{\imath}} = wait$. We will prove:

   $$(pc_{\bar{\imath}} = wait \wedge R) \leadsto (pc_{\bar{\imath}} = crit \wedge R) \leadsto (pc_{\bar{\imath}} = reset\_flag \wedge R) \leadsto$$

   $$\leadsto (pc_i = wait \wedge Enabled(Wait_i)) \leadsto (pc_i = crit).$$

   - $(pc_{\bar{\imath}} = wait \wedge R) \leadsto (pc_{\bar{\imath}} = crit \wedge R)$.
     The only action enabled in this case is $Wait_{\bar{\imath}}$. By definition:

     $$Wait_{\bar{\imath}} \wedge pc_{\bar{\imath}} = wait \wedge R \Rightarrow pc'_{\bar{\imath}} = crit \wedge R',$$

     thus (a) and (b) hold. On the other hand:

     $$Enabled(Wait_{\bar{\imath}}) \equiv pc_{\bar{\imath}} = wait \wedge ((flag_{\bar{\imath}} = 0 \vee turn \neq \bar{\imath}).$$

     We have assumed $pc_{\bar{\imath}} = wait \wedge turn = i$, so it follows:

     $$pc_{\bar{\imath}} = wait \wedge R \Rightarrow Enabled(Wait_{\bar{\imath}}),$$

     thus (c) also holds.
   - $(pc_{\bar{\imath}} = crit \wedge R) \leadsto (pc_{\bar{\imath}} = reset\_flag \wedge R)$.
     The only enabled action is $Crit_{\bar{\imath}}$. By definition:

     $$Crit_{\bar{\imath}} \wedge pc_{\bar{\imath}} = crit \wedge R \Rightarrow pc'_{\bar{\imath}} = reset\_flag \wedge R',$$

     thus (a) and (b) hold. On the other hand:

     $$Enabled(Crit_{\bar{\imath}}) \equiv pc_{\bar{\imath}} = crit.$$

     Therefore we have $pc_{\bar{\imath}} = crit \wedge R \Rightarrow Enabled(Crit_{\bar{\imath}})$, thus (c) holds as well.
   - $(pc_{\bar{\imath}} = reset\_flag \wedge R) \leadsto (pc_i = wait \wedge Enabled(Wait_i))$.
     The only action enabled is $Reset\_flag_{\bar{\imath}}$. By the definition of this action:

     $$Reset\_flag_{\bar{\imath}} \wedge pc_{\bar{\imath}} = reset\_flag \wedge R \Rightarrow$$

     $$\Rightarrow flag'_{\bar{\imath}} = 0 \wedge \mathcal{U}(\{flag_{\bar{\imath}}, pc_{\bar{\imath}}\}),$$

and by the definition of $Enabled(Wait_i)$:

$$pc = wait \wedge flag_{\bar{i}} = 0 \Rightarrow Enabled(Wait_i),$$

thus (a) and (b) hold. On the other hand:

$$pc_{\bar{i}} = reset\_flag \Rightarrow Enabled(Reset\_flag_{\bar{i}}),$$

thus (c) holds too.

- $(pc_i = wait \wedge Enabled(Wait_i)) \leadsto (pc_i = crit)$.
  This was already proven above (see 1.).

Case $pc_{\bar{i}} = crit$**.** We will prove:

$$(pc_{\bar{i}} = crit \wedge R) \leadsto (pc_{\bar{i}} = reset\_flag \wedge R) \leadsto$$

$$\leadsto (pc_i = wait \wedge Enabled(Wait_i)) \leadsto (pc_i = crit).$$

This chain is contained in the chain from the previous case, which is already proven.

Case $pc_{\bar{i}} = reset\_flag$**.** We will prove:

$$(pc_{\bar{i}} = reset\_flag \wedge R) \leadsto$$

$$\leadsto (pc_i = wait \wedge Enabled(Wait_i)) \leadsto (pc_i = crit).$$

This chain is contained in the chain from the previous case, which is already proven.

Case $pc_{\bar{i}} = set\_turn$**.** We will prove:

$$(pc_{\bar{i}} = set\_turn \wedge R) \leadsto$$

$$\leadsto (pc_i = wait \wedge Enabled(Wait_i)) \leadsto (pc_i = crit).$$

This is already proven in (1.ii.).

$\square$

References

[1] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[2] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, California, 1996.

[3] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems. Safety*. Springer-Verlag, New York, 1995.