The assertional versus Tarskian methods

Uri Abraham *

May 12, 2014

Abstract

In this lecture we analyze Peterson's well known two-process critical-section protocol [3] and prove that the mutual exclusion property holds. Our main aim is to explain two approaches by which the correctness of this protocol can be established. The first is the assertional method of proof which is the standard approach to concurrency. This approach uses the notions of global states, steps and histories. The second approach is event-based rather than state-based, and the notion of Tarskian system execution is used in the formalizing of an event-based proof.

1 Mutual exclusion protocol

A critical section is a part of a program that must be executed in isolation from any other process critical-section execution. The critical section problem is to devise a protocol that prevents simultaneous access by two or more processes into their critical sections. For example, a process that sends a file to a printer needs to know that no other process can access the printer during the time of printing. Protocols that guarantee this separation are said to have the mutual exclusion property. An additional property is clearly required: that any process requesting its critical section will be able to access it sooner or later. This desirable "liveness" property is called the lockout freedom property. We will prove here the mutual exclusion property and leave the liveness properties as an exercise.

Before we present Peterson's algorithm in details, we want to discuss mutual exclusion algorithms in general. We follow Lamport's fundamental paper [2].

*Departments of Mathematics and Computer Science, Ben-Gurion University. Models for Concurrency 2014.

Critical-section algorithms involve a number of serial processes, and the structure of the program of each of these concurrently operating processes is the following.

repeat_forever

1. non-critical part (which may last for ever)
2. trying
3. Critical Section (which is assumed to have a finite duration)
4. exit

When a process desires to execute a critical section it begins to execute the trying code. If the execution of the trying part terminates, the process enters its critical section. It is convenient to assume that every execution of the Critical Section part terminates. We assume that the exit part consists of a finite number of instructions whose executions always terminate.

Given an execution of such a critical section algorithm we say that the execution has the Mutual Exclusion Property when for every two Critical Section executions A and B either A precedes B or B precedes A. (Symbolically, A precedes B is written $A < B$.)

2 Peterson's mutual exclusion protocol

The two-process critical section protocol of Peterson is shown in figure 1. The **wait_until** instructions at line 4 is interpreted here as "busy wait". This means that the process (A or B) repeatedly reads into two local variables the Q and *Turn* registers until the required condition holds. The flowcharts of Figure 2 clarifies this, and the proof refers to the flowcharts rather than to the text of figure 1.

Three registers (shared variables) are used by the processes: Q_A, Q_B, and *Turn*. Registers Q_A and Q_B are single-writer boolean registers and are written by processes A and B respectively. The initial value of Q_A and Q_B is *false*. Both processes can write on register *Turn*, which carries values in $\{$*"A"*, *"B"*$\}$, and the initial value of *Turn* is unimportant. The registers are assumed to be serial. Recall that this means that the read/write actions on the registers are assumed to be linearly ordered, and if r is any read action of register R (one of the three registers) then the value returned by

procedure A **repeat_forever** **begin**	**procedure** B **repeat_forever** **begin**
1. non-critical section	1. non-critical section
2. $Q_A := \mathit{true}$;	2. $Q_B := \mathit{true}$;
3. *Turn*:= *"A"*;	3. *Turn*:= *"B"*;
4. **wait_until** $\neg Q_B \vee \mathit{Turn} =$ *"B"*;	4. **wait_until** $\neg Q_A \vee \mathit{Turn} =$ *"A"*;
5. Critical Section of A;	5. Critical Section of B;
6. $Q_A := \mathit{false}$	6. $Q_B := \mathit{false}$
end.	**end**.

Figure 1: Peterson's mutual exclusion protocol. Q_A and Q_B are initially *false*.

r is the value of the rightmost write action on register R that precedes r. (We assume an initial write action that determined the initial value of the register.)

Two local variables are used by process A in the flowchart: r_A is boolean and s_A can hold a value from {*"A"*, *"B"*}. (The corresponding local variables of process B are r_B and s_B.) The nodes of the flowchart are labeled with instructions that the process is about to execute. A directed edge (which we call "arrow") represents the event of executing the instruction labeled at the node that the arrow is leaving. For example arrow $(1_A, 2_A)$ represents writing *true* on Q_A. Nodes 5_A (in the A flowchart) and 5_B (in the B flowchart) are special because they are labeled with a condition (a formula). If the formula is true at a certain state then the arrow labeled with *true* is followed (leading to the critical section) but otherwise the arrow labeled with *false* is taken, which indicates that another round of reading registers is required.

An execution of this protocol is described as a sequence of states that begins with an initial state and such that the passage from one state to the next is caused by an execution of one of the labeled instructions by A or by B. To explain this, we must first define states. For this, two "control variables" are added, CP_A and CP_B. CP_A (Control Position of A) takes

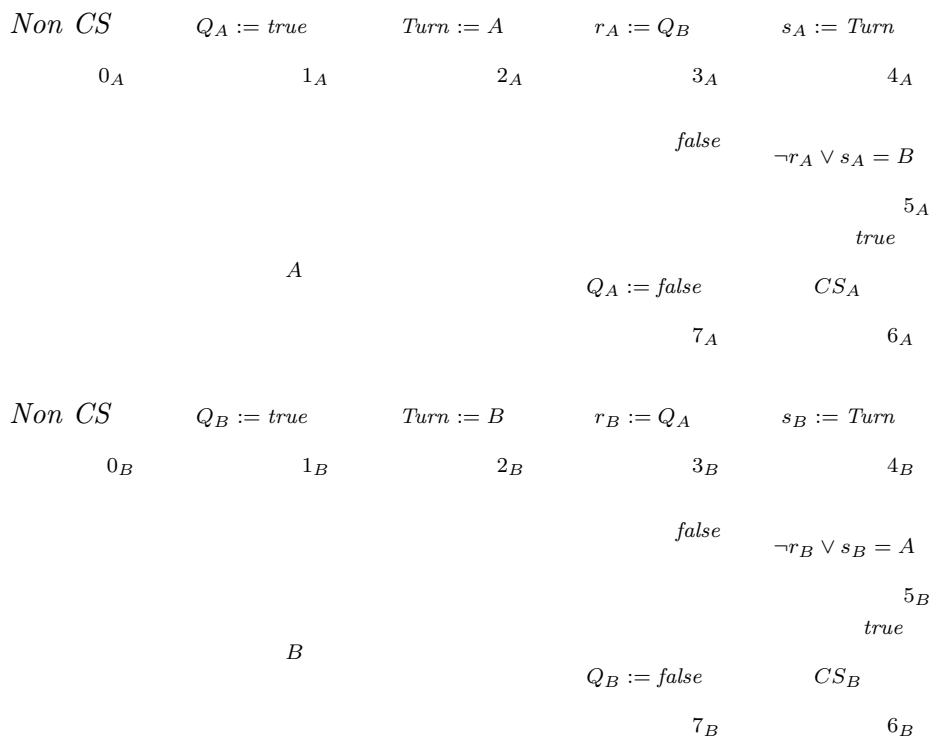

Figure 2: Mutual exclusion flowchart. There are also self-loops not explicitly figured: from 0_A to itself and from 0_B to itself, and likewise from the critical section nodes back to themselves. Variables Q_A and Q_B are initially **false**.

values in $\{0_A, \ldots, 7_A\}$, and CP_B takes values in $\{0_B, \ldots, 7_B\}$.

Definition 2.1 *A state is an assignment σ of a value to each of the state variables*

$$CP_A, CP_B, Q_A, Q_B, \mathit{Turn}, r_A, s_A, r_B, s_B$$

in its range, namely a function σ from the set of state variables such that the following hold.

1. $\sigma(CP_A) \in \{0_A, \ldots, 7_A\}$, $\sigma(CP_B) \in \{0_B, \ldots, 7_B\}$.
2. $Q_A, Q_B \in \{\mathbf{true}, \mathbf{false}\}$,
3. $r_A, r_B, \mathit{Turn} \in \{$ *"A", "B"*$\}$

There are thus $8^2 \times 2^7$ states.

An initial state is a state σ such that $\sigma(CP_A) = 0_A$, $\sigma(CP_B) = 0_B$, $\sigma(Q_A) = \mathit{false}$, and $\sigma(Q_B) = \mathit{false}$. The other values of σ are immaterial. A "step" is a pair of states (σ_1, σ_2) that represent an execution of one of the instructions. A step thus corresponds to an arrow of the flowchart. To each arrow α in the flowchart of A or B we attach a set of steps $\mathit{Stp}(\alpha)$ that represents possible executions corresponding to α.

1. If $\alpha = (0_B, 0_B)$ then $\mathit{Stp}(\alpha)$ is the set of all pairs of states (σ, σ) such that $\sigma(CP_B) = 0_B$. These steps represent the continuation of process B in its stay in the non-critical section.

2. If $\alpha = (0_B, 1_B)$ then $\mathit{Stp}(\alpha)$ is the set of all pairs of states (σ_1, σ_2) such that $\sigma_1(CP_B) = 0_B$, $\sigma_2(CP_B) = 1_B$, and for any other variable v $\sigma_1(v) = \sigma_2(v)$. Informally, we say that a step in $\mathit{Stp}(0_B, 1_B)$ is a "starting step" by B.

3. If $\alpha = (1_B, 2_B)$ then $\mathit{Stp}(\alpha)$ is the set of all pairs of states (σ_1, σ_2) such that $\sigma_1(CP_B) = 1_B$, $\sigma_2(CP_B) = 2_B$, $\sigma_2(Q_B) = \mathit{true}$, but all other variables do not change. Such a step is informally called "writing on Q_B".

4. If $\alpha = (2_B, 3_B)$ then $\mathit{Stp}(\alpha)$ is the set of all pairs of states (σ_1, σ_2) such that $\sigma_1(CP_B) = 2_B$, $\sigma_2(CP_B) = 3_B$, and $\sigma_2(\mathit{Turn}) =$ *"B"*. Other variables do not change. These are the "write on *Turn*" steps by B.

5. If $\alpha = (3_B, 4_B)$, then $\mathit{Stp}(\alpha)$ is the set of all steps (σ_1, σ_2) such that control CP_B moves from 3_B to 4_B, $\sigma_2(r_B) = \sigma_1(Q_A)$ and all other

variables do not change. The effect of reading register Q_A and assigning the value obtained to r_B is thus expressed in the relation $\sigma_2(r_B) = \sigma_1(Q_A)$ holding between the state σ_1 and the resulting state σ_2. Such a step is called "reading of Q_A".

6. Similarly, $Stp(4_B, 5_B)$ steps are defined. We say that these are "reading of *Turn* by B".

7. The steps attached to the *true* and to the *false* edges are defined:

   (a) For $\alpha = (5_B, 6_B)$, the *true* arrow, $Stp(\alpha)$ is the set of all steps (σ_1, σ_2) such that $\sigma_1(CP_B) = 5_B$, $\sigma_1(r_B) = \mathit{false} \vee \sigma_1(s_B) =$ *"A"*, and $\sigma_2(CP_B) = 6_B$; all variables other than CP_B do not change.

   (b) The steps attached to $(5_B, 3_B)$ are similarly defined, except that the condition $\neg r_B \vee s_B = A$ must be evaluated to false at state σ_1.

8. A $(6_B, 6_B)$ step is a pair of states (σ, σ) such that $\sigma(CP_B) = 6_B$. These "CS-stay" steps represent a possible continuation of process B of its stay in the critical section.

9. The $(6_B, 7_B)$ steps corresponding to the arrow leading from 6_B (the CS_2 node) to 7_B change only the control variable of B. A step in $Stp(6_B, 7_B)$ is a "CS exit" step.

10. Finally $Stp(7_B, , 0_B)$ is the set of all steps (σ_1, σ_2) such that $\sigma_1(CP_B) = 7_B$, $\sigma_2(CP_B) = 0_B$, $\sigma_2(Q_B) = \mathit{false}$, and no other variables are changed in these "return" steps.

In a similar vein steps are attached to the edges of the flowchart of A.

Exercise 2.2 *Write down the steps for process A.*

Histories (also called runs) describe executions of the protocol:

Definition 2.3 *A* history *is an infinite sequence of states $\sigma_0, \sigma_1, \ldots$ such that σ_0 is an initial state and for each i (σ_i, σ_{i+1}) is a step in some $Stp(\alpha)$ where α is an arrow either in the flowchart of A or of B.*

"Fairness" is not needed for the mutual-exclusion property, but in order to ensure that every $(0_A, 1_A)$ step (a step in which process A leaves the non critical section) is followed by a successful $(5_A, 6_A)$ step, it is necessary to

assume that process B executes only a finite number of CS-stay steps and that both processes are given sufficiently many steps to execute in order to advance in their algorithm.

The mutual-exclusion property is that in every history there is no state σ_i such that $\sigma_i(CP_A) = 6_A$ and $\sigma_i(CP_B) = 6_B$, namely there is no state in which both processes are in their critical sections.

Definition 2.4 *An* assertion *is a propositional sentence built out of the following atomic propositions (by means of the logical connectives). For a state σ and assertion α, relation $\sigma \models \alpha$ means that α holds (is true) in state σ, or as we say: σ satisfies α.*

1. *If V is a state variable and c some possible value, then $\sigma \models V = c$ iff $\sigma(V) = c$.*
2. *If V is a boolean state variable then $\sigma \models V$ means that $\sigma(V) =$ **true**. For example, $\sigma \models r_A$ means $\sigma(r_A) =$ **true**.*
3. *If V and W are state variables, then $\sigma \models V = W$ means $\sigma(V) = \sigma(W)$.*

If φ is any assertion, then for any state σ, $\sigma \models \varphi$ or $\sigma \not\models \varphi$. That is, σ satisfies φ or negates it. Here is an example of an assertion φ: $(Turn = A) \wedge Q_A$. For any state σ, σ satisfies φ, written symbolically as $\sigma \models \varphi$, iff $\sigma(Turn) =$ *"A"* and $\sigma(Q_A) = true$.

Let MUTEX be the mutual exclusion assertion

$$\neg(CP_A = 6_A \wedge CP_B = 6_B)$$

Our aim is to prove that in every history MUTEX holds. That is, every state in the history satisfies MUTEX. There are other properties of this protocol that one may want to prove. For example the liveness property says that each process can access its critical section if there are sufficiently many steps by this process in the run and if the other process has only finite stays in its critical sections. We shall not deal at this stage with this liveness property.

Proof of the Mutual Exclusion property

A possible approach to proving the mutual exclusion property is to list all states that can appear in any history of the protocol and then to check that none of these states violates the mutual exclusion property. Such a listing

can be obtained by closing the set of initial states under all of the protocol's steps. This "model checking" approach has proved to be of immense importance mainly in industrial applications because it allows for machine correctness checking. Another approach, the one that will be taken here, is to prove the mutual exclusion property not by some exhaustive listing but rather by analyzing the set of steps of the protocol and establishing the invariance of the mutual exclusion property under these steps.

Definition 2.5 *An assertion φ is an* invariant *of the protocol iff every initial state satisfies it, and the following holds for every step (σ, σ'): if σ satisfies φ then σ' satisfies it as well.*

If φ is an invariant, then for every history $\langle \sigma_i \mid i \in \mathbb{N} \rangle$, we have that every σ_i satisfies φ. (Here $\mathbb{N} = \{0, 1, \ldots\}$ is the set of natural numbers.)

Given a statement φ, a proof that it holds in every state of a history is essentially an inductive proof. One proves that it holds at the first state (an initial state) and that if it holds at state σ_i then it also holds at σ_{i+1}. It is often the case in inductive proofs that in order to prove that a certain statement holds we formulate and prove a stronger statement. Here too, in order to prove the mutual exclusion property we shall formulate an assertion φ, prove that it implies MUTEX, and prove that it is an invariant. This invariant will be a conjunction of several simpler invariants which we state separately in the following lemmas.

For easier expression, let's agree to have shorthands such as $CP_A = 0_A, 1_A$ instead of the explicit $CP_A = 0_A \vee CP_A = 1_A$.

Exercise 2.6 *The proofs of Lemmas 2.7 and 2.8 are missing. Please complete the proofs.*

Lemma 2.7 *Each of the following assertions is an invariant.*

1. $CP_A = 2_A, 3_A, 4_A, 5_A, 6_A, 7_A \longrightarrow Q_A$.
2. $CP_A = 0_A, 1_A \longrightarrow \neg Q_A$.
3. $CP_A = 6_A \longrightarrow (\neg r_A \vee s_A = B)$.
4. $CP_B = 2_B, 3_B, 4_B, 5_B, 6_B, 7_B \longrightarrow Q_B$.
5. $CP_B = 0_B, 1_B \longrightarrow \neg Q_B$.
6. $CP_B = 6_B \longrightarrow ((\neg r_B) \vee s_B = A)$.

Proof. Complete the proof here. Directions: you are supposed to prove (1) that each of these assertions holds in any initial state, and (2) that for every step (σ, σ'), for every assertion from this list, if $\sigma \models \varphi$ then $\sigma' \models \varphi$. Take for example the first assertion α, and note that it involves only variables that can be changed by steps of process A. So, if (σ_1, σ_2) is some process B step, then $\sigma_1 \models \alpha$ if and only if $\sigma_2 \models \alpha$. We say that α is "indifferent" with respect to steps by B. Since, α is indifferent for steps by B, we have to check that it is an invariant only for steps by A, since for steps by B this follows trivially. ∎

Lemma 2.8 *The following two statements are invariants.*

1. $[(CP_A = 5_A, 6_A, 7_A) \wedge s_A = B] \longrightarrow Turn = B.$
2. $[(CP_B = 5_B, 6_B, 7_B) \wedge s_B = A] \longrightarrow Turn = A.$

Proof. Complete the proof here. Note that these statements are not indifferent for any of the two processes.

∎

The main invariant needed for the mutual exclusion is presented in the following exercise.

Let u_B, v_B, w_B and symmetrically u_A, v_A, and w_A be short notations for the following assertions (respectively):

$u_B = (\neg Q_B),$	$u_A = (\neg Q_A),$
$v_B = (CP_B = 2_B),$	$v_A = (CP_A = 2_A),$
$w_B = (CP_B = 3_B, 4_B, 5_B) \wedge$ $Turn = B \wedge$ $(CP_B = 5_B \longrightarrow s_B = B) \wedge$ $(CP_B = 4_B, 5_B \longrightarrow r_B).$	$w_A = (CP_A = 3_A, 4_A, 5_A) \wedge$ $Turn = A \wedge$ $(CP_A = 5_A \longrightarrow s_A = A) \wedge$ $(CP_A = 4_A, 5_A \longrightarrow r_A).$

Let ψ_A and ψ_B be the following statements.

ψ_A: $(CP_A = 4_A, 5_A, 6_A) \wedge \neg r_A \longrightarrow (u_B \vee v_B \vee w_B).$

ψ_B: $(CP_B = 4_B, 5_B, 6_B) \wedge \neg r_B \longrightarrow (u_A \vee v_A \vee w_A).$

Note that while the antecedent of ψ_A involves only variables that are local to process A (namely CP_A and r_A) its conclusion involves only variables of B and the two-writer register *Turn*.

Exercise 2.9 *Let δ be the conjunction of the invariants of the two previous lemmas.*

Define ϵ to be the following statement: $\delta \wedge \psi_A \wedge \psi_B$. Prove that ϵ is an invariant.

Hints for the proof. First check that every initial state satisfies ϵ. Since you have already shown that δ is an invariant (Exercises 2.7, 2.8) it suffices for this part to prove that ψ_A and ψ_B hold in every initial state. (In fact it is possible to show this only for ψ_A, say, and to comment that the proof for ψ_B is symmetrically obtained.)

Then you are supposed to consider all possible steps (S, T), assume that state S satisfies the invariant ϵ, and conclude that state T satisfies it as well.

Now, if we interchange A and B in assertion ϵ we get ϵ again, and if we interchange A and B in the definition of the steps of A then we get the steps of B. It follows from this symmetry that it suffices to prove that ϵ is invariant under the steps of A (and then to argue that the proof for the steps of B is obtained by interchanging A with B).

Steps of the form (S, S) are trivial, and so we have to look only at steps of A of the nine forms (S, T) in which $S \neq T$. We assume that $S \models \epsilon$ and have to prove that $T \models \epsilon$.

Since δ is an invariant, the assumption that it holds in S implies that it holds in T, and hence we may assume in the following proof that both S and T satisfy δ.

Assume that the step is by process A. There are 9 types of steps and you should consider each of them. Steps in $Stp(0_A, 1_A)$, steps in $Stp(1_A, 2_A)$ etc. We give two examples.

1. Suppose that (S, T) is some $(2_A, 3_A)$ step. Since $T(CP_A) = 3_A$, $T \models CP_A \neq 4_A, 5_A, 6_A$. Hence the antecedent of ψ_A is false in T, and hence $T \models \psi_A$. We will prove that $T \models \psi_B$ by observing that $M \models w_A$. Since $M \models CP_A = 3_A \wedge \mathit{Turn} = A$, this is quite evident.

2. Suppose that step (S, T) is in $Stp(3_A, 4_A)$. By definition of this step
$$T(r_A) = S(Q_B). \tag{1}$$
We have to prove that (a) $T \models \psi_A$ and (b) $T \models \psi_B$.

   (a) Assume that T satisfies the antecedent of ψ_A, and in particular that $T \models \neg r_A$. Conclude from (1) that $S(Q_B) = \mathbf{false}$. Hence also $T(Q_B) = False$ (because steps by A do not change the value of local variables of B). But u_B is $\neg Q_B$. So $T \models u_B$, and hence $T \models \psi_A$.

(b) Now assume that T satisfies the antecedent of ψ_B. Since this antecedent is indifferent to steps by A, S satisfies this antecedent as well, and since S satisfies ψ_B, $S \models (u_A \vee v_A \vee w_A)$. It is not the case that $S \models u_A$, since u_A is $\neg Q_A$ (and $S(CP_A) = 3_A$ implies $S(Q_A) = \mathbf{true}$ by invariant δ). For a similar reason it is not the case that $S \models v_A$. Hence necessarily $S \models w_A$. That is, S satisfies the conjunction of

i. $CP_A = 3_A, 4_A, 5_A$ and
ii. *Turn*=A and
iii. $CP_A = 5_A \longrightarrow s_A = A$ and
iv. $CP_A = 4_A, 5_A \longrightarrow r_A$.

But T also satisfies this conjunction. T satisfies item i since (S, T) is a $(3_A, 4_A)$ step. T satisfies item ii since this step didn't change the value of *Turn*. T satisfies iii since the premise of this implication is false. And finally T satisfies item iv by the following argument. We are assuming that T satisfies the antecedent of ψ_B, and hence that S satisfies it too, and in particular $S \models CP_B = 4_B, 5_B, 6_B$. By invariant δ, this implies that $S \models Q_B$ (item 4 of Lemma 2.7). By (1), this implies that $T(r_A) = \mathbf{true}$. That is, the consequence of implication iv holds in T, and hence item iv holds in T. So $T \models \psi_B$ as required.

The final step in the correctness proof consists in proving that our invariant ϵ implies MUTEX. For a contradiction assume that MUTEX does not hold in some state σ despite the fact that it satisfies ϵ. By Lemma 2.7, σ satisfies

$$(\neg r_A \vee s_A = B) \ \mathbf{and} \ (\neg r_B \vee s_B = A). \tag{2}$$

In case $\neg r_A$ holds in σ (a similar argument can be given in case $\neg r_B$) the antecedent of ψ_A holds in σ, and the fact that ψ_A holds in σ implies that $u_B \vee v_B \vee w_B$ holds there as well. Now u_B is $\neg Q_B$ and this cannot be true in σ since $\sigma(CP_B) = 6_B$ and the invariant of Lemma 2.7(3) (which is part of δ) tells that $\sigma \models Q_B$. Also, v_B is $CP_B = 2_B$, which clearly is false in σ, because $CP_B = 6_B$ is assumed at σ. So w_B remains as the sole disjunct that can hold at σ, which implies that $CP_B = 3_B, 4_B, 5_B$, and this is again a contradiction.

So now assume that r_A and r_B are **true** at σ. Hence $s_A = B$ and $s_B = A$ (by formula (2)). Then by the two invariants at Lemma 2.8, both $Turn = B$

and $Turn = A$ in state σ, which is surely impossible, and this contradiction proves the mutual exclusion property.

3 Why Peterson's algorithm is much simpler in textbooks?

Peterson's algorithm is one of the most popular distributed algorithms, and it appears in almost every textbook and course that deals with concurrency.

4 Peterson's algorithm, an event based approach

In the previous lecture we approached Peterson's mutual exclusion protocol from a *state based* point of view. We used states, steps and histories to describe the working of the algorithm. The proof of the mutual exclusion property was *assertional*, that is (essentially) an inductive proof in which the truth of an *invariant statement* is shown to hold in each of the states of a run. In the present lecture we describe an *event based* approach to Peterson's protocol. At first we give an informal but intuitive proof for the mutual exclusion property and then we discuss the question of formality and ask what is needed in order to transform that informal argument into a mathematical proof. (The notion of Tarskian system executions plays a role in this transformation.)

We intend to prove the mutual-exclusion property. For this, we have to prove that if cs_1 and cs_2 are two critical-section events then either $cs_1 < cs_2$ or else $cs_2 < cs_1$. That is, the two critical-section events cannot be concurrent. If both critical-section events are by the same process then this is evident since each process is serial and its events are linearly ordered. So we may assume that they are by different processes: one is by A and the other by B.

Analyzing the algorithm for process A (the flowchart in Figure 2) we can realize (informally at this stage) that any critical section event E by process A is preceded by the following sequence of events. (And a symmetric analysis can be done for process B.)

1. A write of value **true** on register Q_A corresponding to node 1_A with its "$Q_A := \mathbf{true}$" instruction. This event is denoted $Write_{Q_A}(E)$.

2. A write on register *Turn* of value *"A"* follows $Write_{Q_A}(E)$. This write event is denoted $Write_{Turn}(E)$.

3. Then follows a sequence of read events of registers Q_B and *Turn* corresponding to nodes 3_A and 4_A of the flowchart. Since E is a critical-section event, condition $\neg r_A \vee s_A = B$ holds before control passed from 5_A to 6_A, and hence there has to be at least one "successful" read. That is, a read of register Q_B that returned the value **false** or a read of register *Turn* that returned the value *"B"*. There are possibly both a successful read of Q_B and a successful read of *Turn*, and there may be several non-successful reads before E, but we single-out a successful read and denote it with $sr(E)$.

These events of process A are ordered as follows:

$$Write_{Q_A}(E) < Write_{Turn}(E) < sr(E) < E. \tag{3}$$

And if E is a critical section event by process B then similarly we have

$$Write_{Q_B}(E) < Write_{Turn}(E) < sr(E) < E. \tag{4}$$

We intend to prove a somewhat more interesting property than the mutual-exclusion: we will prove that if E_1 and E_2 are critical section executions then their ordering is determined by the ordering of events $Write_{Turn}(E_1)$ and $Write_{Turn}(E_2)$. That is if $Write_{Turn}(E_1) < Write_{Turn}(E_2)$ then we also have that $E_1 < E_2$. Since register *Turn* is assumed to be serial, we either have $Write_{Turn}(E_1) < Write_{Turn}(E_2)$ or else $Write_{Turn}(E_2) < Write_{Turn}(E_1)$ and hence $E_1 < E_2$ or $E_2 < E_1$ follows which is the mutual-exclusion property that we want to prove.

We give an informal proof first of this ordering property. Assume that

$$Write_{Turn}(E_1) < Write_{Turn}(E_2) \tag{5}$$

and suppose for example that E_1 is by process A and E_2 by B. So $Turn(E_1)$ is a write of *"A"* and $Turn(E_2)$ a write of *"B"*. Consider $sr(E_2)$ the successful read in E_2. There are two possibilities (looking at the flowchart of B): $sr(E_2)$ can be a read of register Q_A that returned **false**, and it can also be a read of *Turn* that returned *"A"*. We have two different arguments for each of these cases, but the conclusion is in both cases that $E_1 < E_2$.

Case 1: The successful read that precedes E_2 is a read of register Q_A that returns false. Assumption (5) and the ordering equations (3) and (4) yields

$$Write_{Q_A}(E_1) < Write_{Turn}(E_1) < Write_{Turn}(E_2) < sr(E_2)$$

and hence $Write_{Q_A}(E_1) < sr(E_2)$ follows. But as $Write_{Q_A}(E_1)$ is a write of value **true**, and $sr(E_2)$ is a read of value **false**, there has to be a write W (of value **false**) on register Q_A such that $Write_{Q_A}(E) < W < sr(E_2)$. Only process A can write on Q_A and since there is no write by process A between $Write_{Q_A}(E_1)$ and E_1, it must be the case that $E_1 < W$. Hence $E_1 < W < sr(E_2) < E_2$, and $E_1 < E_2$ follows.

Case 2: $r = sr(E_2)$ is a successful read of *Turn* of value *"A"*. Since (by item 5) $Write_{Turn}(E_1) < Write_{Turn}(E_2) < r$, and as $Write_{Turn}(E_2)$ is a write of value *"B"* and the successful read r is of value *"A"*, there has to be a write event V of value *"A"* in between $Write_{Turn}(E_2)$ and r. Hence (as $Write_{Turn}(E_1) < Write_{Turn}(E_2)$) $Write_{Turn}(E_1) < V$. Yet only process A can write *"A"* on *Turn*, and since A contains no write event on *Turn* between $Write_{Turn}(E_1)$ and E_1, we necessarily have that $E_1 < V$. Hence $E_1 < V < r < E_2$ shows that $E_1 < E_2$ as required.

Whenever I read this proof, I accompany my reading with some illustrations, and I strongly advice the student to use diagrams that show the arrangement of the events on a time axis. Diagrams and informal arguments have their place in mathematics, and the possibility of arguing informally is very important in both engineering and science. Formal proofs however are equally important since only a formal proof can provide a guarantee that the argument has no gap and the result can be trusted. The best is to have the possibility of moving swiftly across different levels of formality within a comprehensive framework. In the following section we present a formal correctness proof for the algorithm of Peterson that resembles very much the informal proof given above.

5 A formal behavioral proof of Peterson's algorithm

What is the main difference between a formal and an informal proof? What is missing from the correctness proof of the algorithm of Peterson that we gave above and in what sense is this proof informal? We will try to answer these questions, and this answer will open the way to a formal correctness proof of the algorithm that follows the same intuitions and structure that the informal proof had.

We first want to clarify that by "mathematical proof" we do not mean what is meant in mathematical logic when this term is defined. In mathe-

matical logic courses we learn that a proof is a finite sequence of symbols that obeys a certain rule. For example, a sequence of formulas so that each formula in the sequence is either a logical axiom or a a result of applying the modus ponens rule to two previous formulas in the sequence. This is *not* what we intend here by the term mathematical proof. This purely syntactical notion of a proof is important of course both in theory and in applications, but it is not what mathematician usually do when they prove theorems, and it is not what computer scientists usually want to check when they seek correctness proofs for their distributed algorithms. Usually, mathematicians consider a certain family of structures and when they claim that a certain sentence φ is a theorem they prove that each structure in that family satisfies φ. The proof that indeed each structure satisfies φ can be more or less rigorous but the framework is clear: for a formal proof one needs the following well defined objects:

1. A language L, namely a collection of symbols (a signature) with formation rules that define what are the well-formed formulas of the language.

2. A notion of "interpretation", that is of a structure that interprets L.

3. A "satisfaction" relation which defines mathematically when a formula φ is satisfied (or falsified) in a structure.

I think that the dividing line between an informal and formal proof is not that the formal proof is more detailed and more careful in checking all possible cases, but that the formal proof has a well defined notion of what are the structures to which the proof refers to.

So the main features that were missing from the intuitive correctness proof given above for the Peterson's algorithm are the exact definitions of the language and structures that can sustain the proof. We never defined the structures that arise when the algorithm is executed, and so we have left it to the reader's imagination and experience to figure out the objects to which the different terms of the proof refer to. In the first part of this article we did define executions of the algorithm as history sequences. That is, as sequences of states that represent possible executions of the algorithm. A history sequence was very suitable for an assertional correctness proof, but not for the second proof. That second proof was based on the events rather than states and it employed several functions such as $Write_{Turn}$ that took CS events into write events. We cannot deny that a history sequence is a structure, but it is not a structure that can support such functions. What

we need is a way to transform any history sequence into a corresponding Tarskian system execution that can support a proof in the style that was given in section 4. Conceptually, this transformation is easy: take a history sequence and add, on top of it, all functions that are needed. Probably, this is exactly what the reader had done when reading that proof, but it turns out that to carry out this obvious idea requires careful work and some not so obvious decisions must be made in order to choose between different alternatives.

We divide the following presentation into two parts. Firstly, we shall take that intuitive proof of section 4 and try to explicate it by defining its language and underlying structures, and then we shall ask in an exercise to reproduce the proof of section 4 in that formally defined language and its structures. It should come as no surprise to learn that a system execution language and Tarskian structures will be employed in this explication.

Then, in the second part of the presentation we shall define how Tarskian system executions of the algorithm of Peterson can be defined. Putting together these parts yields a formal correctness proof. A long procedure, but one that is guided by intuition.

The L_P language for Peterson's algorithm

If we analyze the arguments for the mutual exclusion property given above in section 4 for the Peterson's algorithm we find immediately that the main objects of that discussion are the read/write events and the critical section executions. We also see that the ordering of events within each protocol execution plays an important role. For example we used the fact that the write in register Q_A of the value **true** is followed by a write in register *Turn* of the value *"A"*. We also used the assumption that the registers are serial, so that the read/write events related to a register are linearly ordered and each read gets the value of the last write on that register that preceded it. I shall define now a language (called L_P) and its interpreting structures within which our correctness proof can be conducted. We begin by defining the signature of the language (which is a list of predicates, function symbols and constants). The reader may want to return to the lectures in which the general notion of system execution languages and structures were defined. The list is in Figure 3.

Examples of statements that can be expressed in the L_P language.

1. *Plain English*: Event e_1 is the successor of e_0 in the sequence of P_A events.

predicates	arity	Sort	intended meaning
Write	1	*Event*	$Write(e)$ says e is a write event.
Read	1	*Event*	$Read(e)$ says e is a read event.
cs	1	*Event*	$cs(x)$ says x is a critical section event.
P_A, P_B	1	*Event*	$P_A(e)$ says event e is by process P_A.
$<$	2	$Event \times Event$	The precedence relation.
functions			
Address	1	$Event \to \{Q_A, Q_B, Turn\}$	$Address(e)$ is the register accessed by event e.
Val	1	$Event \to Data$	$Val(e)$ is the value of read/write e.
Data sort constants			
Q_A, Q_B, *Turn*			names of registers,
"A", "B", **true**, **false**			four distinct tokens.

Figure 3: The signature of the L_P language.

in L_P: $P_A(e_0) \wedge P_A(e_1) \wedge (e_0 < e_1) \wedge \neg\exists x(P_A(x) \wedge e_0 < x \wedge x < e_1)$.

2. *Plain English:* every *cs* event by P_A has a successor that is a write on Q_A of the value *false*. (This statement may be true or false; we only want here to formally write it.)

   in LP: Write the formal sentence.

3. *Plain English:* Mutual Exclusion Property: if $e_1 \neq e_2$ are two critical-section events then either $e_1 < e_2$ or $e_2 < e_1$. *in LP*: Write a formal expression of the Mutual Exclusion property.

5.1 Properties of executions

When Peterson's Algorithm is executed, the resulting events satisfy certain properties that are "intuitively true" and which can be expressed in the L_P language defined above. We call these properties "axioms" because we view them as statements that express basic properties of executions of the algorithm. These statements have however a meaning that is independent of the algorithm itself, and they can be studied (and should be studied) without any reference to the protocol. Namely, we would like to obtain their consequences, which can be derived without the knowledge that they are properties of the protocol's executions. We shall prove that the mutual exclusion property ME is one of the consequences of these axioms, and this will

convince us that Peterson's Algorithm satisfies the mutual exclusion property. At a later stage we shall outline the proof that indeed any execution of Peterson's algorithm generates a structure that satisfies these axioms.

The Peterson's Axiom List is divided into four parts P1–P4 (P4 is divided into P4A and P4B) as follows:

P1. Temporal precedence relation: Relation $<$ on the events is a partial ordering that satisfies the Russell–Wiener property and Lamport's finiteness property. (See the lectures on system executions and time.)

P2. Register Axioms. Every write event in register Q_A is by process P_A. Any write event in register Q_B is by process P_B. Each of the three registers Q_A, Q_B and *Turn* is serial. For every register R from those three, we have that:

P2.1 The set of read/write events e such that $Address(e) = R$ is serially ordered. Here R is Q_A, Q_B, or *Turn*.

P2.2 There exist an initial write events on every register. The initial write events on registers Q_A and Q_B are of value *false*. Every read r of register R returns the value of the last write on R that precedes r.

P3. Seriality of processes:

P3.1 Every event is either in process P_A or in P_B.

P3.2 The processes are serial: all events of P_A (or P_B) are serially ordered: $\forall a_1, a_2(P_A(a_1) \wedge P_A(a_2) \wedge (a_1 \neq a_2) \rightarrow a_1 < a_2 \vee E_2 < a_1)$.

P4A. Protocol's execution by P_A. If E is any *cs* event by process P_A, then there are events $Write_{Q_A}(E)$, $Write_{Turn}(E)$, $\mathrm{sr}(E)$ (all by P_A) such that the following hold.

P4A.0
$$Write_{Q_A}(E) < Write_{Turn}(E) < \mathrm{sr}(E) < E.$$

P4A.1 $Write_{Q_A}(E)$ is a write into register Q_A of value *true* and is such that there is no write W on register Q_A such that $Write_{Q_A}(E) < W < E$.

P4A.2 $Write_{Turn}(E)$ is a write into register *Turn*, of value *"A"* and such that there is no write V onto register *Turn* by process P_A with $Write_{Turn} < V < E$. Moreover, any write on register *Turn* of value *"A"* is by process P_A.

P4A.3 $\mathrm{sr}(E)$ is either a read of register Q_B of value *false*, or a read of register *Turn* of value *"B"*.

There exists a write onto register Q_A of value *false*, and it is the first event by P_A that comes after E.

P4B. Protocol's execution by P_B. If E is any *cs* event by process P_B, then there are events $Write_{Q_B}(E)$, $Write_{Turn}(E)$, $\mathrm{sr}(E)$ (all by P_B) such that the following hold.

P4B.0

$$Write_{Q_B}(E) < Write_{Turn}(E) < \mathrm{sr}(E) < E.$$

P4B.1 $Write_{Q_B}(E)$ is a write into register Q_B of value *true* and is such that there is no write W on register Q_B such that $Write_{Q_B}(E) < W < E$.

P4B.2 $Write_{Turn}(E)$ is a write into register *Turn*, of value *"B"* and such that there is no write V onto register *Turn* by process P_B with $Write_{Turn} < V < E$. Moreover, any write in register *Turn* of value *"B"* is by process P_B.

P4B.3 $\mathrm{sr}(E)$ is either a read of register Q_A of value *false*, or a read of register *Turn* of value *"A"*.

There exists a write onto register Q_B of value *false*, and it is the first event by P_B that comes after E.

Theorem 5.1 *The LP Axiom List imply the Mutual Exclusion Property. In fact, if E_1 and E_2 are critical section events such that $Write_{Turn}(E_1) < Write_{Turn}(E_2)$, then $E_1 < E_2$.*

Exercise 5.2 *Prove this theorem. (Prove also that the mutual exclusion property is a consequence of the theorem's statement.)*

6 Executions of the algorithm generate Tarskian system executions

We have defined in the previous section a family of Tarskian system executions (those system executions that satisfy properties P1–P4) and we have proved that in any such structure the mutual exclusion property holds. We did not, however, connect the text of the algorithm (the flowchart of figure 2) with these system executions, and hence a main item is so far missing from our correctness proof. What is missing is a definition of a family $\mathcal{F}$ of system executions such that the following hold:

1. In a well-defined, formal sense $\mathcal{F}$ is the set of all possible executions of the algorithm.

2. If $M \in \mathcal{F}$ then M satisfies properties P1–P4.

There is an obvious candidate for a set $\mathcal{F}$ of all possible executions of the algorithm, namely the set of all histories as defined in 2.3. The problem with this candidate is that we do not (yet) have a well-defined mathematical definition of a satisfaction relation $H \models \phi$ which can tell us in a formal way that H satisfies properties P1–P4. We know what it means that a state statement ϵ holds in a state S, and we know how to prove that ϵ is an invariant. It is by proving that ϵ is an invariant that we were able to prove that it holds in every state of a history sequence, and it is by means of invariants that we were able to show that the algorithm satisfies the mutual-exclusion property. There however an obvious difference between a state assertion such as ϵ and the properties that were expressed in P1–P4. The latter are not properties of a states but of an execution. Property P1 for example is a quantification statement about events. It has the form "for every event E there are events W etc."

It is possible (and not difficult) to take a history $H = \langle \sigma_i \mid i \in \omega \rangle$, and to consider the step executions $e_i = (\sigma_i, \sigma_{i+1})$ as the main members of a new structure for which a statement such as P4 is meaningful. We will take a different road though which can be explained by means of the following observation[1]. We discern two types of properties among P1–P4. First we have properties that are general and not specifically related to the algorithm at hand. The register axioms for example are an assumption about the communication means (namely about the serial registers employed). We do not expect to prove that Q_A for example is serial by any reference to the

[1]This direction is a main contribution of [1].

predicates	Sort
Write	*Event*
Read	*Event*
cs	*Event*
$<$	$Event \times Event$
functions	
Address	$Event \to \{Q_A, Q_B, Turn\}$
Val	$Event \to Data$
Data sort constants	
Q_A, Q_B, *Turn*	register names
"A", "B", **true**, **false**	tokens.

Figure 4: The signature of the L_PA language.

algorithm itself. It is an assumption on which the correction proof relies. It is in properties P4A and P4B that the algorithm is manifested. But here we observe something very simple which is the key for our definition of $\mathcal{F}$. Property P4A refers only to the algorithm of process P_A, and P4B only to P_B. In other words, P4A is a property of executions of a serial algorithm (the algorithm of P_A) that would be true even if P_B does not exists and the read actions return arbitrary values (in the range of the registers). The language in which the properties of P4A are expressed is called L_PA. This language is described in Figure 4; it is a sublanguage of the L_P language described in Figure 3.

Since the properties of P4A do not involve process P_B, their proof requires only runs in which process P_A act alone: there is no need to look at complex histories in which the two processes interleave. This point is so simple that we must explain it in full details, else the student may fail to grasp it–being so trivial.

In figure 5 we have reproduced the algorithm of process P_A, but now we view it as running alone without any assumption of a process P_B that runs concurrently and without any assumption of serial registers that the process read and write. If so, you may ask, what are the values that reads of registers Q_B and *Turn* return? And what is the effect of a write on a non-existent register? The answer is that reads of a register can return an arbitrary value (in the range of the register), and write actions have no effect what so ever. This is evident from the following definition of the steps.

There are only three state-variables of the stand-alone P_A algorithm: CP_A, r_A, s_A. A stand-alone P_A state is a function S that gives to each of

procedure P_A

0_A non-critical section;

1_A $Q_A := true$;

2_A $Turn$:= "A";

3_A $r_A := Q_B$;

4_A $s_A := Turn$;

5_A **if** $r_A \wedge s_A =$ "A" **goto** 3_A;

6_A Critical Section of A;

7_A $Q_A := false$

8_A **goto** 0_A;

Figure 5: The P_A algorithm running in isolation.

these three state variables a value in their range: $S(CP_A) \in \{0_A - 8_A\}$, $S(r_A) \in \{true, false\}$, and $S(s_A) \in \{\text{“}A\text{”}, \text{“}B\text{”}\}$. An initial state is one in which the value of PC_A is 0_A. In the following definition of P_A stand alone steps we refer to P_A stand alone states and state variables.

Definition 6.1 *The following set of pairs is the set of P_A stand alone steps.*

1. A $(0_A, 0_A)$ step is a pair of states $p = (S, T)$ such that $S = T$ and $S(CP_A) = 0_A$. Such a step is said to be *external*.

2. A $(0_A, 1_A)$ step is a pair of states (S, T) such that $S(CP_A) = 0_A$ and $T(CP_A) = 1_A$, but $T(x) = S(x)$ for any of the other two state variables.

3. A $(1_A, 2_A)$ step is a pair of states $p = (S, T)$ such that $S(CP_A) = 1_A$ and $T(CP_A) = 2_A$. For any state variable other then CP_A, S and T have the same value. We say that this step is "a write on register Q_A of value **true**". This saying is just a predicate of that step (a description of the step), and no value is actually written anywhere.

4. A $(2_A, 3_A)$ step is a pair of states (S, T) such that $S(CP_A) = 2_A$,

$T(CP_A) = 3_A$, and $T(x) = S(x)$ for any other variable x. We say that this step is "a write on register *Turn* of the value *"A"*."

5. A $(3_A, 4_A)$ step is a pair of states (S, T) such that $S(CP_A) = 3_A$, $T(CP_A) = 4_A$, and $T(s_A) = S(s_A)$. So $T(r_A)$ can have any value v (that is allowed for this variable, that is $v \in \{\mathbf{true}, \mathbf{false}\}$). We say that this step is "a read of register Q_B" and we also say that "the value of this step is v". So the value of this step is $T(r_A)$.

6. A $(4_A, 5_A)$ step is a pair of states (S, T) such that $S(CP_A) = 4_A$, $T(CP_A) = 5_A$, and $T(r_A) = S(r_A)$. The value $T(s_A) \in \{$ *"A"*, *"B"*$\}$ is arbitrary, and is said to be the value of this step which is said to be a "read of register *Turn*" step.

7. A $(5_A, 3_A)$ step is a pair of states (S, T) such that $S(CP_A) = 5_A$, $T(CP_A) = 3_A$, $T(x) = S(x)$ for any other state variable, and $S \models r_A \wedge s_A =$ *"A"*.

8. A $(5_A, 6_A)$ step is a pair of states (S, T) such that $S(CP_A) = 5_A$, $T(CP_A) = 6_A$, $T(x) = S(x)$ for any other state variable, and $S \models \neg(r_A \wedge s_A =$ *"A"*$)$. This step is said to be a "CS entry" step.

9. A $(6_A, 7_A)$ step is a pair of states (S, T) such that $S(CP_A) = 6_A$, $T(CP_A) = 7_A$ and for any other variable $T(x) = S(x)$. This is a "CS exit" step. The critical section event extends from the entry and up to the subsequent exit step.

10. A $(7_A, 8_A)$ step is a pair of states (S, T) such that $S(CP_A) = 7_A$, $T(CP_A) = 8_A$, and $T(x) = S(x)$ for the other two variables. We say that this step is a "write on register Q_A of the value **false**".

11. A $(8_A, 0_A)$ step is a pair of states (S, T) such that $S(CP_A) = 8_A$, $T(CP_A) = 0_A$, and $T(x) = S(x)$ for the other variables. This is a "back to the non-critical-section" step.

A stand-alone history of P_A is an infinite sequence $H = \langle S_i \mid i \in \omega \rangle$ (where $\omega = 0, 1, \ldots$ is the set of natural numbers) or a finite sequence $H = \langle S_i \mid i \leq j \rangle$ of P_A stand alone states S_i such that S_0 is an initial state and every pair of consecutive states (S_i, S_{i+1}) is a P_A stand-alone step.

Now we transform H into a Tarskian structure denoted $\overline{H}$, and called the "history structure" of H, as follows. The basic idea is to make the step occurrences in H to become the members (events) of $\overline{H}$.

Definition 6.2 *Definition of the derived structure $\overline{H}$ from the finite history H.*

1. The universe of $\overline{H}$ consists of two sorts. *Events*, and atemporal values. The events of $\overline{H}$ form an abstract set $Event = \{e_i \mid i \in \omega\}$ (or $Event = \{e_i \mid i < j\}$ in case H has S_j as last state). Event e_i is associated with the step (S_i, S_{i+1}). We also have the sort of atemporal values in $\overline{H}$ which contains the following sixteen elements.

$$\{\mathbf{true}, \mathbf{false}, \text{“}A\text{”}, \text{“}B\text{”}, Q_A, Q_B, \mathit{Turn}, 0_A, \ldots 8_A\}.$$

2. A precedence relation on the events is defined in $\overline{H}$: $e_m < e_n$ iff $m < n$.

3. Predicates CS_{entry}, CS_{exit}, *Read*, *Write* are defined as follows on the events of $\overline{H}$. $CS_{entry}(e_i)$ holds in $\overline{H}$ iff (S_i, S_{i+1}) is a $(5_A, 6_A)$ step. $CS_{exit}(e_i)$ holds in $\overline{H}$ iff (S_i, S_{i+1}) is a $(6_A, 7_A)$ step. $\mathit{Read}(e_i)$ holds iff (S_i, S_{i+1}) is one of the two reading steps $(3_A, 4_A)$ and $(4_A, 5_A)$. $\mathit{Write}(e_i)$ holds iff (S_i, S_{i+1}) is one of the three writing steps $(1_A, 2_A)$, $(2_A, 3_A)$, $(7_A, 8_A)$.

4. The *Address* and *Val* functions are defined over the *Read*/ *Write* events. For such an event e, $\mathit{Address}(e)$ is the register on which e operates. If e is associated to a $(1_A, 2_A)$ step, then $\mathit{Address}(e) = Q_A$, and $\mathit{Val}(e) =$ **true**. If e is associated to a $(2_A, 3_A)$ step then $\mathit{Address}(e) = \mathit{Turn}$ and $\mathit{Val}(e) =$ *“A”*. If e is associated to a $(3_A, 4_A)$ step (S, T), then $\mathit{Address}(e) = Q_B$, and $\mathit{Val}(e) = T(r_A)$. If e is associated to a $(4_A, 5_A)$ step (S, T) then $\mathit{Address}(e) = \mathit{Turn}$ and $\mathit{Val}(e) = T(s_A)$. If e is a $(7_A, 8_A)$ step (S, T) then $\mathit{Address}(e) = Q_A$ and $\mathit{Val}(e) =$ **false**.

This ends the definition of the Tarskian system execution $\overline{H}$ derived from H, a stand alone history of process P_A. $\overline{H}$ is an interpretation of the language L_PA described in Figure 4 which is a sublanguage of the L_P language described in Figure 3.

At first, it appears quite obvious that $\overline{H}$ satisfies property P4A: the first read/write event in any execution of the algorithm is a write of value **true** on register Q_A, and this event is followed by a write of *“A”* on register *Turn*; then there is a sequence of read events of registers Q_B and *Turn* until a successful read is obtained which allows entry to the critical section. Since any critical section event is obtained via such a sequence of events, it is obvious that property P4A holds. If we want to find a formal proof however we soon discover that it is by no means an obvious task. One thing should

If E is any *cs* event, then there are events $Write_{Q_A}(E)$, $Write_{Turn}(E)$, $\mathrm{sr}(E)$ such that the following hold.

$$Write_{Q_A}(E) < Write_{Turn}(E) < \mathrm{sr}(E) < E.$$

P4.1 $Write_{Q_A}(E)$ is a write into register Q_A of value *true* and is such that there is no write W on register Q_A such that $Write_{Q_A}(E) < W < E$.

P4.2 $Write_{Turn}(E)$ is a write into register *Turn*, of value *"A"* and such that there is no write V onto register *Turn* by process P_A with $Write_{Turn} < V < E$. Moreover, any write on register *Turn* by process P_A is of value *"A"*.

P4.3 $\mathrm{sr}(E)$ is either a read of register Q_B of value *false*, or a read of register *Turn* of value *"B"*.

Figure 6: The P_A stand alone properties P4A.

be clear: that such a proof has to use the definition of steps that was given above since it is this definition that expresses the semantics of the programs whose proof we seek.

The pair $\mathcal{H} = (H, \overline{H})$ where H is a finite P_A stand alone history, $\overline{H}$ is the resulting history structure is a "compound state" which replaces the notion of state in the following discussion. If $H = \langle S_i \mid i \leq j \rangle$ then the function f that carries $e_i \in \overline{H}$ to the corresponding step (S_i, S_{i+1}) is also part of $\mathcal{H}$, but since it is definable we do not always mention f explicitly. In fact Since H alone determines both $\overline{H}$ and f, we can say that $\mathcal{H}$ is the pair *derived from* H.

Our aim is to prove that the properties displayed in figure 6 hold in $\overline{H}$, whenever H is a history sequence of the stand alone P_A process. To achieve that, we shall to define a statement η which implies P4A and which is an invariant. But this couldn't be literally so of course since P4A (Figure 6) is not a statement about states. So we have to extend somewhat the realm of statements which our states can support. Given any $i \in \omega$ let $H_i = \langle S_j \mid j \leq i \rangle$ be the initial segment of our history H containing the first $i + 1$ states, and let $\overline{H}_i$ be the substructure of $\overline{H}$ generated by H_i. So the set of events of $\overline{H}_i$ is $\{e_j \mid j < i\}$ (and in particular $\overline{H}_0 = \emptyset$). We say that S_i is the last state of $\overline{H}_i$. For $i > 0$, e_{i-1} is the last event of $\overline{H}_i$, and the step that corresponds to this event is the pair (S_{i-1}, S_i). The main observation is that $\overline{H}_{i+1}$ is obtained from $\overline{H}_i$ by application of step (S_i, S_{i+1}), that is, by adding event e_i which corresponds to this step.

Let H be a finite P_A stand alone history and let $\mathcal{H} = (H, \overline{H}))$ be the derived pair. Let S be the last state of H; then we can also say that S is the last state of $\mathcal{H}$. We are interested in statements of the form $\alpha \to \beta$ where α is a propositional state formula and β is an L_PA sentence. We define

$$\mathcal{H} \models \alpha \to \beta \text{ if } S \not\models \alpha \text{ or } \overline{H} \models \beta. \tag{6}$$

We say that $\alpha \to \beta$ is a basic compound sentence. We are interested in sentences that are conjunctions of basic compound sentences.

Let $\mathcal{H} = (H, \overline{H})$ be a compound state derived from H and suppose that S is its last state. Let (S, T) be a P_A stand alone step, and let H' be the history obtained from H by adding T as a last state, and let $\overline{H'}$ be the history structure derived from H'. Define $\mathcal{H}' = (H', \overline{H'})$. Then the pair $\langle \mathcal{H}, \mathcal{H}') \rangle$ is said to be an *extended step.*

Now we can be more precise in the description of the proof of P4A. We will define a statement $\alpha \to \beta$ and prove by induction on i that $(\overline{H_i}, H_i, S_i) \models \alpha \to \beta$. In fact we will prove that $\alpha \to \beta$ is an invariant in the following sense. If $\langle (\mathcal{H}, \mathcal{H}')$ is an extended step and $\mathcal{H} \models \alpha \to \beta$, then $\mathcal{H}' \models \alpha \to \beta$.

We define the following formulas.
$\varphi_0(u)$ is the formula

$$\mathit{Write}(u) \wedge \mathit{Address}(u) = Q_A \wedge \mathit{Val}(u) = \mathbf{true} \wedge \forall w(u < w \wedge \mathit{Write}(w) \to \mathit{Address}(w) \neq Q_A).$$

$\varphi_1(v)$ is the formula

$$\mathit{Write}(v) \wedge \mathit{Address}(v) = \mathit{Turn} \wedge \mathit{Val}(v) = \text{“}A\text{”} \wedge \forall w(v < w \to \neg \mathit{Write}(w))$$

.
$\psi_0(r)$ is the formula $\mathit{Read}(r) \wedge \mathit{Address}(r) = Q_B \wedge \mathit{Val}(r) = \mathbf{false}$.
$\psi_1(s)$ is the formula $\mathit{Read}(s) \wedge \mathit{Address}(s) = \mathit{Turn} \wedge \mathit{Val}(s) = \text{“}B\text{”}$.

We can define now our extended variant. It is the conjunction of the following compound statements 6.1–6.4.

6.1 $CP_A = 2_A \Rightarrow \exists u(\varphi_0(u))$.

6.2 $CP_A = 3_A, 4_A, 5_A, 6_A \Rightarrow \exists u \exists v(u < v \wedge \varphi_0(u) \wedge \varphi_1(v))$.

6.3 $CP_A = 4_A, 5_A, 6_A \wedge r_A = \mathbf{false} \Rightarrow \exists u \exists v \exists r(u < v < r \wedge \psi_0(r) \wedge \varphi_0(u) \wedge \varphi_1(v))$.

6.4 $CP_A = 5_A, 6_A \wedge s_A = \text{“}B\text{”} \Rightarrow \exists u \exists v \exists s(u < v < s \wedge \psi_1(s) \wedge \varphi_0(u) \wedge \varphi_1(v))$.

Implication 6.ℓ is denoted $\alpha_\ell \Rightarrow \beta_\ell$ for $\ell = 1, \ldots, 4$.

Exercise 6.3 *Prove that the conjunction of the four implications is an invariant.*

Directions. You have to consider in turn each one of the P_A stand alone steps (S, T) (Definition 6.1). For each such step you have to assume that H is some arbitrary finite history sequence of stand alone P_A states that ends with state S, and let H' be the sequence obtained by adding (concatenating) T to H. Then let $\overline{H}$ and $\overline{H}'$ be the L_PA structures derived from H and H' respectively. So $\overline{H}'$ contains one event added at the end of $\overline{H}$, namely its last event which corresponds to step (S, T). Define $\mathcal{H} = (H, \overline{H})$ and $\mathcal{H}' = (H', \overline{H}')$. You assume that $\mathcal{H} \models \alpha \Rightarrow \beta$ for each of the four implications $\alpha \Rightarrow \beta$ above, and then prove that $\mathcal{H}' \models \alpha \Rightarrow \beta$ as well.

For example, suppose that we deal with (S, T) that is a $(5_A, 3_A)$ step. We assume that $\mathcal{H}$ satisfies each $\alpha_\ell \Rightarrow \beta_\ell$ and we have to prove that $\mathcal{H}'$ also satisfies them. Let's see for example how we can prove that $\mathcal{H}'$ satisfies $\alpha_2 \Rightarrow \beta_2$. This means (see 6) that if state T (which is the last state of $\mathcal{H}'$) satisfies α_2, then we have to prove that $\overline{H}'$ satisfies β_2. In our case, $T \models \alpha_2$, since α_2 is the disjunction $CP_A = 3_A, 4_A, 5_A, 6_A$, and since (S, T) is a $(5_A, 3_A)$ step (so that $T \models CP_A = 3_A$). So we have to prove that $\mathcal{H}' \models \exists u \exists v (u < v \wedge \varphi_0(u) \wedge \varphi_1(v))$. Now, as $S \models CP_A = 5_A$, we clearly have that $S \models \alpha_2$ and by assumption on $\mathcal{H}$, $\overline{H} \models \beta_2$. So there are in $\overline{H}$ events u_0 and v_0 such that

$$\overline{H} \models (u_0 < v_0 \wedge \varphi_0(u_0) \wedge \varphi_1(v_0)). \tag{7}$$

We must prove the following

Claim 6.4 $\overline{H}' \models (u_0 < v_0 \wedge \varphi_0(u_0) \wedge \varphi_1(v_0))$.

Proof outline. What is the difference between the two structures $\overline{H}$ and $\overline{H}'$? $\overline{H}$ is a substructure of $\overline{H}'$, and $\overline{H}'$ is obtained by adding one event to $\overline{H}$: d the last event of $\overline{H}'$ is such that $x < d$ for every event x of $\overline{H}$. Moreover, since (S, T) is a$(5_A, 3_A)$ step $S \models r_A \wedge s_A =$ *"A"* (and $T \models r_A \wedge s_A =$ *"A"* since these variables do not change in this step). As d corresponds to this step, predicates *Read* and *Write* do not apply to event d (see the definition of the derived structure, Definition 6.2). Since the ordering on the events of $\overline{H}$ remain the same in the extension $\overline{H}'$, $u_0 < v_0$ also holds in $\overline{H}'$. So there are two items that we have to prove in order to prove Claim 6.4.

1. $\overline{H}' \models \varphi_0(u_0)$. That is, $\overline{H}' \models \mathit{Write}(u_0) \wedge \mathit{Address}(u_0) = Q_A \wedge \mathit{Val}(u_0) = \mathbf{true} \wedge \forall w (u_0 < w \wedge \mathit{Write}(w) \rightarrow \mathit{Address}(w) \neq Q_A)$. Since

$\overline{H}$ is a substructure of $\overline{H}'$, and as (7) holds, it immediately follows from

$$\overline{H} \models \mathit{Write}(u_0) \wedge \mathit{Address}(u_0) = Q_A \wedge \mathit{Val}(u_0) = \mathbf{true}$$

that

$$\overline{H}' \models \mathit{Write}(u_0) \wedge \mathit{Address}(u_0) = Q_A \wedge \mathit{Val}(u_0) = \mathbf{true}.$$

Thus it remains to prove that $\overline{H}' \models \forall w(u_0 < w \wedge \mathit{Write}(w) \rightarrow \mathit{Address}(w) \neq Q_A)$. So let w be an arbitrary event in $\overline{H}'$ such that $u_0 < w$ and $\mathit{Write}(w)$. Since predicate *Write* does not apply to d, $w \neq d$ and so w is a member of the substructure $\overline{H}$. Hence $\overline{H} \models \mathit{Address}(w) \neq Q_A$, and this implies that $\overline{H}' \models \mathit{Address}(w) \neq Q_A$ as well, which settles item 1.

2. $\overline{H}' \models \varphi_0(v_0)$. We leave this proof to the reader.

Note on substructures. The proof outline that was given above relies on the notion of substructure. We recall the definition of substructure. (In our lectures on Tarskian structures in "Basic logic and model theory", and specifically in Section 2.1, we defined this notion for single sorted structures.) Let M and N be two (multisorted) structures that interpret the same language L. We say that M is a substructure of N if $|M| \subseteq |N|$. That is, the universe of M is a subset of the universe of N and the following holds:

1. For every sort S of the language L, $S^M = S^N \cap |M|$. In other words, for any member a of the universe of M, a is in sort S in M if and only if a is in sort S in N.

2. For every predicate P of the language L of arity k, for every k-tuple $\overline{a}$ of elements of the universe of M, $\overline{a} \in P^M$ iff $\overline{a} \in P^N$.

3. For every function symbol F of the language of arity k and for every k-tuple $\overline{a}$ of elements of the universe of M, $F^M(\overline{a}) = F^N(\overline{a})$ (and in particular, if $\overline{a}$ in a k-tuple of members of the universe of M then $F^N(\overline{a})$ is in the universe of M).

It follows from this definition that if $\varphi(x_1, \ldots, x_n)$ is any quantifier free formula of the language (that is a formula that involves no quantifiers) then for any n-tuple $\overline{a}$ of members of the universe of M, $M \models \varphi[\overline{a}]$ iff $M \models \varphi[\overline{a}]$.

We are now able to conclude the event based correctness proof of the Peterson two-process mutual exclusion algorithm[2]. Suppose the processes P_A and P_B that execute the algorithm of Peterson, but in an environment in which there is no restriction whatsoever on the operations of the diverse registers used (except that a read of a register returns a value that is one of the values that that register is designed to carry). So in this modeling mode a write operation has no effect (or has arbitrary effect, which is the same) and a read operation returns arbitrary values (chosen from the range of values of the register read). What would be the structures that model these *unrestricted* executions? These models would be Tarskian system executions that interpret the language L_P and that satisfy properties P1, P3, P4A and P4B. Let $calM$ be the set of all structures that interpret the L_P language, and are such that:

1. The restriction of M to the events of P_A is a derived history structure of some P_A stand alone history H (that is, this restriction is $\overline{H}$), and the restriction of M to the events of P_B is a derived history structure of some P_B stand alone history.

2. The ordering of the events $<^M$ is a linear ordering when restricted to the P_A events, and a linear ordering when restricted to P_B, but the possible relations between events of the two processes is arbitrary– provided that it satisfies the Russell–Wiener property and the Lamport's finiteness property.

For example, if we take any derived history $\overline{H_A}$ and a derive history $\overline{H_B}$ of processes P_A and P_B and we decide that the events of the two processes interleave, an events of P_A is followed by an events of P_B and an event of P_B is followed by an event of P_A, then we get a linear ordering of the events which results in a structure in $\mathcal{M}$. We have seen in details that property P4A holds whenever the algorithm of P_A is executed alone, and a symmetric proof would show that property P4B holds. Thus if M is a structure in $\mathcal{M}$ then properties P1, P3, P4 hold. Let $\mathcal{M}$ be the set of all such unrestricted executions. We do not expect that the mutual exclusion property holds in every structure of $\mathcal{M}$. But if we let $\mathcal{M}_{serial}$ be the subset of $\mathcal{M}$ consisting of those structures in which the registers are serial (this is the set of *restricted structures*) then any structure M in $\mathcal{M}_{serial}$ satisfies all properties $P1 - -P4$ and hence satisfies the mutual exclusion property as well. $\mathcal{M}_{serial}$ is the set of all executions of the Peterson algorithm in which

[2]We follow here a central idea of [1], the distinction between nonrestricted and restricted semantics of concurrency.

the registers behave correctly, and thus the proof shows that if the processes follow their algorithms and the registers are serial, then the mutual exclusion property is guaranteed.

Our experience has shown that what we have seen here in our exploration of the Peterson algorithm is a general phenomenon. The semantics of serial processes that operate concurrently can be split into two parts. The first part is in analyzing each process in isolation as if the communication between the different processes is completely arbitrary. In this analysis properties of stand alone executions of each process are derived, and when they are combined with the assumed properties of the communication devices the correctness of the algorithm can be proved.

References

[1] U. Abraham. *Models for Concurrency.* Gordon and Breach, 1999.

[2] Leslie Lamport. 1986. The mutual exclusion problem: part I - a theory of interprocess communication. J. ACM 33, 2 (April 1986), 313-326. partII - statement and solutions. 327-348.

[3] G.L. Peterson. Myths About the Mutual Exclusion Problem, *Information Processing Letters* 12(3) 1981, 115-116)