Peterson's Mutual Exclusion Algorithm as Feedback Control

Arjun Sanjeev, Venkatesh Choppella, and Viswanath Kasturi

Abstract We present a feedback control approach to the problem of mutual exclusion, using transition systems. As an example to illustrate the idea, we consider Peterson's mutual exclusion algorithm and model the artefacts used in the algorithm as transition systems, the appropriate feedback composition of which yields the desired mutual exclusion property to the resultant system. As we see later, the solution we build is very modular, and the dynamics can be explained with simple equational reasoning.

1 Introduction

We look at control theory and how we can apply the same to problems in concurrency. It is a very popular design principle in engineering systems. In the control approach, we employ additional actors along with the existing components so that they restrict the components in some well-defined manner to achieve the target behaviour. These additional actors are called *controllers*. In this context, a controller issues *commands* that may allow a process to move to a new state according to its natural dynamics, or force the process to stay in the same state. For example, a process which wants to execute its critical section may not be allowed by the controller to do so, possibly because of some other process executing the critical section. At the same time, the controller should also take care it does not put too much restriction on the processes. It is this approach that we wish to explore in this paper.

We describe the mutual exclusion problem using a novel method of formally specifying any given problem. The objective of this paper is to use feedback control to solve the problem of mutual exclusion. Feedback control is a very popular design principle which forms the foundation of much of

IIIT Hyderabad, Telangana, India

engineering science. In this work, we model all actors in the form of transition systems. We employ a centralised hub controller that issues control inputs to the processes, thus controlling their behaviour. This becomes feedback control as the controller also gets continuous knowledge about the behaviour of the processes in the form of inputs. Thus, the state update of the processes becomes completely dependant on the feedback loop.

In our model, the first and the foremost task is to get the architecture right. The ingenuity of the approach lies in how the various components are composed with each other. After wiring the actors together using appropriate logic, we go ahead and define the dynamics of each block, including the *control law*[1].

The most important advantage of the compositional approach is to enable us to think of the system in a modular way, emphasising the interfaces between the components and how they are interconnected. The modularity in architecture also makes the proofs modular.

Apart from modularity, another important property of composition of systems is that of the assume-guarantee[5]. Each component follows the assume-guarantee that *if* it is provided with inputs that satisfy certain properties, *then* its outputs will also satisfy certain other properties. It is interesting to note that when two components A and B are composed in our model, A implicitly discharges B's assumptions on its inputs as A's outputs satisfy those assumptions. Thus, the composition of A and B together satisfies the properties satisfied by B's outputs when it is a standalone component.

To demonstrate the effectiveness of the idea, we consider Peterson's algorithm[12]. The reason why we choose the algorithm over other mutual exclusion algorithms is the fact that it has a very simple structure, and a relatively easy proof. We model the algorithm using feedback composition of various systems and prove its correctness using the dynamics. As we will see later, it is a good example where the dynamics require the knowledge of not just the current state of the processes, but also depends on the derivative in the sense that it depends on two consecutive values.

2 Transition Systems

A transition system (TS) is a six-tuple $\langle X, X^0, U, f, Y, h\rangle$. The symbols can be explained as follows:

1. X is a set called the state space. 2. X^0 is a subset of X called the set of initial states. 3. U is a finite set called the set of inputs. 4. $\longrightarrow\ \subseteq X \times U \times X$ is called the transition relation. If $(x, u, x') \in\longrightarrow$ we shall write $x \xrightarrow{u} x'$.

[1] The transition function of the controller is called the 'control law'.

5. Y is a set called the output space. 6. $h : X \longrightarrow Y$ is a map called the output map (or 'view').

We assume that there are no blocked states, i.e., for every $x \in X$, $\exists\, u \in U$, $x' \in X$ such that $x \xrightarrow{u} x'$. When U is a singleton we write $U = \{*\}$ and in place of $x \xrightarrow{*} x'$ we simply write $x \longrightarrow x'$. When the system is transparent its state becomes its output. In other words, $Y = X$ and $h = \mathcal{I}_X$ (identity function on X).

2.1 Runs and Traces

We briefly describe the concepts of runs, fair runs and traces. A run is an infinite sequence of the form

$$x_0 \xrightarrow{u_0} x_1 \xrightarrow{u_1} x_2 \xrightarrow{u_2} \ldots$$

where $x_i \in X$ for all i and $x_0 \in X^0$. A run is said to be fair if each $u \in U$ occurs infinitely often in the run.

The trace of the above run is the infinite sequence $(h(x_0), h(x_1), h(x_2), \ldots)$. A trace is said to be fair if it is the output of a fair run.

2.2 Specification

We express the specification of a problem in the following terms.

We are given a set $\mathcal{Z}$ as the observable space. Let $\mathcal{Z}^\omega$ denote the space of infinite sequences with entries from $\mathcal{Z}$.

Problem specification: Let a subset $\mathcal{O}$ of $\mathcal{Z}^\omega$ be given. Construct a non-trivial transition system with output space $Y = \mathcal{Z}$ such that the set of fair traces is contained in $\mathcal{O}$.

We do not define the word 'non-trivial', as in any problem a trivial solution can be recognised easily.

3 Mutual Exclusion

In concurrent programming, a critical section is the part of a program that accesses shared resource(s). The property of mutual exclusion requires that no two concurrent processes execute their critical sections at the same time, and is done to prevent race conditions.

In this section, we model a process in the form of a transition system, describe informally the conditions to be satisfied for mutual exclusion, followed by a formal specification of the problem.

3.1 Process as Transition System

With respect to the critical section, each process can be in one of four states:

Thinking The process is not interested in the critical section, and is working on its own outside the critical section.
Hungry The process wants to enter the critical section, but it is still outside the critical section.
Eating The process is executing the critical section.
Full [2] The process is exiting the critical section after execution, but it is still inside the critical section.

The four states - thinking (t), hungry (h), eating (e) and full (f) - form the abstraction over the actual state of a process, and can be called the *activity* of the process. We define the activity set Act as $Act = \{t, h, e, f\}$. Now we can define a process as a transition system as described below:

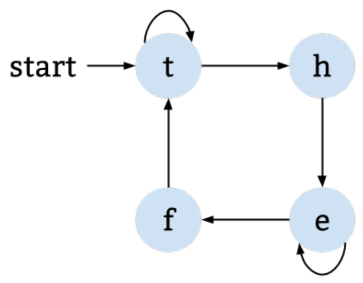

Fig. 1 Process states and transitions.

1. $X = Act = \{t, h, e, f\}$.
2. $X^0 = \{t\}$. 3. $U = \{*\}$.
4. $\longrightarrow$ is defined as $x \to next(x)$, where $next : Act \longrightarrow Act$ is defined by the following transitions: (fig. 1)
   $next(t) = t$ or h, $next(h) = e$, $next(e) = e$ or f, $next(f) = t$
5. $Y = X$. 6. $h = \mathcal{I}_X$.

3.2 Problem Statement

We are interested in the case of two concurrent processes accessing shared resource(s). Each process evolves by its own iteration. We want to impose the constraint of mutual exclusion on their co-evolution by requiring that not both of them are in their critical sections at the same time.

We must also ensure that the constraint is not overly restrictive. This means that two properties essentially need to be satisfied:

[2] In the context of the problem of dining philosophers, which is a more general version of the problem of mutual exclusion, it is customary to have only the three states thinking, hungry, and eating. It seems logical to us to add the fourth state 'full' for the sake of convenience, as we will see later.

1. No deadlock: There should be no situation where neither process can move forever, one which may arise when each process waits for a resource held by each other.
2. No starvation: If a process wants to enter the critical section, it should not be denied the access because of the other process having higher priority.

To avoid the situation where a process waiting for the shared resources get blocked infinitely, we make an assumption that once a process enters its critical section, it should leave the section in finite time.

3.3 Formal Specification of the Problem

Let $\mathcal{Z}^\omega = (Act \times Act)^\omega$ be the set of all infinite sequences with entries from $\mathcal{Z} = Act \times Act$, the observable space. Consider two concurrent processes P_A and P_B, whose states are x_A and x_B, respectively.

The assumption that a process stays in the critical section only for a finite time can be written as follows:

- If $x_A^m \in \{e, f\}$ for some m, then $\exists\ n > m$ such that $x_A^n = t$.
- If $x_B^m \in \{e, f\}$ for some m, then $\exists\ n > m$ such that $x_B^n = t$.

The problem is to construct a non-trivial transition system with output space $Y = \mathcal{Z} = Act \times Act$, such that the set of all fair traces is contained in $\mathcal{O} \subseteq \mathcal{Z}^\omega$. The set $\mathcal{O}$ is such that the traces in $\mathcal{O}$ satisfy the following conditions:

C1 $(x_A^0, x_B^0) = (t, t)$.
This is the 'initial' condition.

C2 For any $m > 0$ such that $(x_A^m, x_B^m) \neq (t, t)$, $\exists\ n > m$ such that $(x_A^n, x_B^n) \neq (x_A^m, x_B^m)$.
This is the 'no deadlock' condition.

C3 For any $m > 0$ such that $x_A^m = h$, $\exists\ n > m$ such that $x_A^n = e$ (same for x_B).
This is the 'no starvation' condition. This is a special case of a 'progress' (or 'liveness') condition.

C4 $(x_A^n, x_B^n) \notin \{e, f\} \times \{e, f\}$ for all $n > 0$.
This is the 'mutual exclusion' condition. This is a special case of a 'safety' condition.

4 Peterson's Algorithm

In simple terms, Peterson's algorithm can be explained as follows. When a process wants to enter the critical section, it makes its intention public by setting its *flag* variable. At the same time, it also courteously gives priority to the other process by setting the *turn* variable to the other process. A process can enter its critical section if the other process does not want to enter its critical section, or if the latter has given priority to the former to do so. The processes unset their flag variables after exiting from the critical section.

Consider two concurrent processes P_A and P_B under execution. We want to make sure that not both P_A and P_B are in their critical sections at any given time. As in the original version of the algorithm, we have two flag variables - F_A and F_B - and one turn variable - T. The processes have write access to the respective flags and the turn variable. In order to solve the mutual exclusion problem by using Peterson's algorithm, we model both the systems, as well as the flag and turn variables as transition systems, and connect them together with a hub controller H. The controller reads the states of the flags and turn, and sends binary control inputs to the processes - 0 to stay in the same state, and 1 to switch to the next state - hence controlling the behaviour of the processes, while also ensuring mutual exclusion.

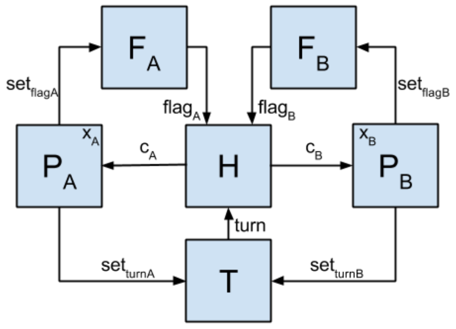

Fig. 2 The interconnect between the processes and the variables, along with the hub controller.

4.1 Components as Subsystems

In this section, we model different components of the algorithm as subsystems and describe how the subsystems are connected to each other.

4.1.1 The processes

The process system P_K (where $K \in \{A, B\}$) takes the output of hub H as input, and sends its outputs to F_K and T.

Its state is a pair (x_{old}, x_{new}), where x_{new} is the actual current state of the process, and x_{old} is its state in the previous iteration. This is done so that it can send appropriate values to F_K and T depending on its state change. If the state changes from $t \longrightarrow h$ (i.e., it wants to enter the CS), it sets its flag and turn by sending 1 to both. If the state changes from $e \longrightarrow f$ (i.e., exiting the CS), it unsets its flag by sending 0 to F_K. In all other cases, it sends $\perp$ (to do nothing) to the systems. It receives binary control input from H: 0 to stay in the same state, 1 to switch to the next state.

1. $X = Act \times Act$. 2. $X^0 = \{(t,t)\}$. 3. $U = \mathbb{B}$.
4. $\longrightarrow$ takes the state (x_{old}, x_{new}), control input c, and computes the new state (x'_{old}, x'_{new}) as follows:
$$(x_{old}, x_{new}) \xrightarrow{c} (x'_{old}, x'_{new})$$
where
$$\begin{aligned}(x'_{old}, x'_{new}) &= (x_{new}, next(x_{new})), \quad \text{if } c = 1\\ &= (x_{new}, x_{new}), \quad \text{otherwise}\end{aligned}$$
Note: According to the above, we define a function g_K (where $K \in \{A, B\}$) as follows:
$$x'_K = g_K(x_K, c_K)$$
where x_K is the actual state of a process P_K, and c_K is the control input.
5. $Y = \{0, 1, \perp\} \times \{1, \perp\}$.
6. h is the view function that is defined as given below:
$$\begin{aligned}h(x_{old}, x_{new}) &= (set_{flag}, set_{turn})\\ &= (1,1), \quad \text{if } x_{new} = h \wedge x_{old} = t\\ &= (0,\perp), \quad \text{if } x_{new} = f \wedge x_{old} = e\\ &= (\perp,\perp), \quad \text{otherwise}\end{aligned}$$
Note: According to the above, we define a function d_K (where $K \in \{A, B\}$) as follows:
$$(set'_{flag\,K}, set'_{turn\,K}) = d_K(x_K, x'_K)$$
where $set_{flag\,K}$ is the signal to F_K from P_K to set/unset the flag, and $set_{turn\,K}$ is the signal to T to set the turn.

4.1.2 The flag variables

The flag system F_K (where $K \in \{A, B\}$) takes the output of P_K as input, and sends its output to H.

It has binary state: 1 if P_K wants to enter CS, 0 otherwise. It can receive 0 (to unset the flag), 1 (to set the flag) or $\perp$ (to do nothing) from P_K as input. Its output is the same as its state.

1. $X = \mathbb{B}$. 2. $X^0 = \{0\}$. 3. $U = \{0, 1, \perp\}$.
4. $\longrightarrow$ takes the state $flag$, input u, and computes the new state $flag'$ as follows:

$$flag \xrightarrow{u} flag'$$

where

$$\begin{aligned} flag' &= flag, \quad \text{if } u = \perp \\ &= u, \quad \text{otherwise} \end{aligned}$$

Note: According to the above, we define a function f_K (where $K \in \{A, B\}$) as follows:

$$flag'_K = f_K(flag_K, set'_{flag\,K})$$

where $flag_K$ is the state of F_K (i.e., the flag), and $set_{flag\,K}$ is the signal to F_K from P_K to set/unset the flag.

5. $Y = X$. 6. $h = \mathcal{I}_X$.

4.1.3 The turn variable

The turn system T takes the outputs of P_A and P_B as inputs, and sends its output to H.

Its state is one among A or B, indicating which among P_A and P_B has priority. Its initial state is set to A, but it is irrelevant and can as well be B. It receives 1 (to give priority to the other process) or $\perp$ (to do nothing) from each process. If it receives 1 from only one process, the priority goes to the other process. If both processes send 1, by design, we make sure that only one will be considered, thus forcing strict alternation. ($A \oplus B$ selects A or B depending on the system's hardware.) If both processes send $\perp$, the state stays the same. Its output is the same as its state.

1. $X = \{A, B\}$. 2. $X^0 = \{A\}$. 3. $U = \{1, \perp\} \times \{1, \perp\}$.
4. $\longrightarrow$ takes the state *turn*, inputs (set_A, set_B), and computes the new state *turn'* as follows:

$$turn \xrightarrow{(set_A, set_B)} turn'$$

where

$$\begin{aligned} turn' &= A \oplus B, \quad \text{if } set_A = 1 \wedge set_B = 1 \\ &= B, \quad \text{if } set_A = 1 \\ &= A, \quad \text{if } set_B = 1 \\ &= turn, \quad \text{otherwise} \end{aligned}$$

Note: According to the above, we define a function t as follows:

$$turn' = tn(turn, set'_{turn\,A}, set'_{turn\,B})$$

where $turn$ is the state of T (i.e., the turn), and $set_{turn\,A}$ and $set_{turn\,B}$ are the set signals to T from P_A and P_B, respectively.

5. $Y = X$. 6. $h = \mathcal{I}_X$.

4.1.4 The hub controller

The hub controller system H takes the outputs of F_A, F_B and T as inputs, and sends the control inputs to the P_A and P_B.

Its state is a binary pair, which are the control inputs to P_A and P_B. By receiving the binary flags from F_A and F_B, as well as the turn from T, it decides control inputs: 0 to force the process to stay in the same state, and 1 to allow to move to the next state. Its output is the same as its state.

1. $X = \mathbb{B} \times \mathbb{B}$. 2. $X^0 = \{(1,1)\}$. 3. $U = \mathbb{B} \times \mathbb{B} \times \{A, B\}$.
4. $\longrightarrow$ takes the state (c_A, c_B), inputs $(flag_A, flag_B, turn)$, and computes the new state (c'_A, c'_B) as follows:
$$(c_A, c_B) \xrightarrow{(flag_A, flag_B, turn)} (c'_A, c'_B)$$
where
$$\begin{aligned} c'_A &= 0, \quad \text{if } flag_A = 1 \wedge flag_B = 1 \wedge turn = B \\ &= 1, \quad \text{otherwise} \\ c'_B &= 0, \quad \text{if } flag_B = 1 \wedge flag_A = 1 \wedge turn = A \\ &= 1, \quad \text{otherwise} \end{aligned}$$
Note: According to the above, we define a function h as follows:
$$(c_A, c_B) = hc(flag_A, flag_B, turn)$$
where $turn$ is the state of T (i.e., the turn), and $flag_A$ and $flag_B$ are the states of F_A and F_B (i.e., the flags), respectively.
5. $Y = X$. 6. $h = \mathcal{I}_X$.

4.2 Dynamics of the Composite System

Let x_A and x_B be the states of processes P_A and P_B, respectively. Let $set_{flag\,A}$ and $set_{flag\,B}$ be the signals to flags sent by the processes P_A and P_B, respectively, and let $flag_A$ and $flag_B$ be the outputs from the flag systems F_A and F_B. Let $set_{turn\,A}$ and $set_{turn\,B}$ be the set signals sent by the processes P_A and P_B to turn system T, respectively. Let $turn$ be the output of T which goes to the controller H. Let c_A and c_B be the control inputs sent by the controller H to the processes P_A and P_B, respectively. All labelling is done in fig. 2.

Interconnecting all the systems (P_A, P_B, F_A, F_B, T and H) with each other, we get the following equations that describe the dynamics of the composite system:

$$\begin{aligned} x_A^0 = t,\ x_B^0 = t,\ set^0_{flag\,A} = \bot,\ set^0_{flag\,B} = \bot \\ set^0_{turn\,A} = \bot,\ set^0_{turn\,B} = \bot,\ turn^0 = A \\ flag_A^0 = 0,\ flag_B^0 = 0 \end{aligned}$$

$$(c_A, c_B) = hc(flag_A, flag_B, turn) \tag{1}$$
$$x'_A = g_A(x_A, c_A) \tag{2}$$
$$x'_B = g_B(x_B, c_B) \tag{3}$$
$$(set'_{flag\,A}, set'_{turn\,A}) = d_A(x_A, x'_A) \tag{4}$$
$$(set'_{flag\,B}, set'_{turn\,B}) = d_B(x_B, x'_B) \tag{5}$$
$$flag'_A = f_A(flag_A, set'_{flag\,A}) \tag{6}$$
$$flag'_B = f_B(flag_B, set'_{flag\,B}) \tag{7}$$
$$turn' = tn(turn, set'_{turn\,A}, set'_{turn\,B}) \tag{8}$$

All variables are initialised with values depending on the system initialisations. The hub controller moves first by computing the pair of control inputs (c_A, c_B) by using $flag_A$, $flag_B$ and $turn$ which it receives as inputs (eq. (1)). P_A and P_B move next by changing their states according to eq. (2) and eq. (3), thus computing x'_A and x'_B, respectively. Based on the updated states x'_A and x'_B, the processes compute the signals $set'_{flag\,A}$ and $set'_{flag\,B}$ to set/unset the flag variables, and $set'_{turn\,A}$ and $set'_{turn\,B}$ to set the turn variable using eq. (4) and eq. (5). After receiving the respective signals, $flag'_A$, $flag'_B$ and $turn'$ are computed by systems F_A, F_B and T, using eq. (6), eq. (7) and eq. (8), respectively. The system T updates its state to $turn'$ and sends the same to the controller H. The flag systems F_A and F_B also send their updated states to the controller H, and the process continues.

4.3 Proof of Correctness

Let $\overline{X}$ denote the state space of the composite system, i.e.,

$$\overline{X} = P_A.X \times P_B.X \times F_A.X \times F_B.X \times T.X \times H.X$$

We are skipping the input space of the composite system as it is too cumbersome to describe, and is left for the reader to figure out.

Let the state of each system be represented as follows:

$$P_A.x = (x_{A\,old}, x_{A\,new}),\ P_B.x = (x_{B\,old}, x_{B\,new}),$$
$$T.x = turn,\ \ F_A.x = flag_A,\ \ F_B.x = flag_B,$$
$$H.x = (c_A, c_B)$$

We define the observation map $\tau : \overline{X} \longrightarrow Act \times Act$ as

$\tau((x_{A\,old}, x_{A\,new}), (x_{B\,old}, x_{B\,new}), flag_A, flag_B, turn, (c_A, c_B)) = (x_{A\,new}, x_{B\,new})$

Let $\mathcal{T}$ denote the set of all fair traces of $\overline{X}$ starting with $((t,t),(t,t),1,1,A,(1,1))$. To prove the solution, we need to show that the traces in $\mathcal{T}$ map into $\mathcal{O}$, i.e., satisfy the formal specifications mentioned.

In this paper, we just briefly prove the mutual exclusion property, i.e., condition **C4**, and give short sketches of the proofs of other conditions. We have verified the correctness of the model using TLC, the TLA+ model checker. The complete proof will be done in an upcoming thesis.

Before we move to the proof, we prove the following lemma:

Lemma 1. *For $K \in \{A, B\}$, $x_K^n \in \{h, e\} \iff flag_K^n = 1$.*

Proof. As is seen from the process dynamics, the flag variable of a process gets set and unset only when the process moves from t to h and e to f, respectively. Hence, when the process is in state h or e, its flag will be 1 and when it is in state t or f, the flag will be 0.

Theorem 1 (C1). $(x_A^0, x_B^0) = (t, t)$.

Proof. This is true because of the way we have initialised the traces.

Theorem 2 (C2). *For any $m > 0$ such that $(x_A^m, x_B^m) \neq (t, t)$, $\exists\ n > m$ such that $(x_A^n, x_B^n) \neq (x_A^m, x_B^m)$.*

Proof. An intuitive reasoning about the proof is that at least one among c_A and c_B is set to 1 at any time in the execution (from eq. (1)), which means that at least one among processes P_A and P_B always moves using *next*. Hence, the state changes as follows: h to e and f to t in a single step (according to fig. 1), and e to t in a finite number of steps (due to assumption).

Theorem 3 (C3). *For any $m > 0$ such that $x_A^m = h$, $\exists\ n > m$ such that $x_A^n = e$ (same for x_B).*

Proof. Consider the process P_A. Let $x_A^m = h$, which means $flag_A^m = 1$.

If $x_B^m \in \{t, f\} \implies flag_B^m = 0$ (from lemma 1) $\implies c_A^m = 1$ (from eq. (1)) $\implies x_A^{m+1} = e$ (from eq. (2)).

If $x_B^m = e \implies \exists\ n > m$ such that $x_B^n = t$, where n is the smallest such number (from assumption) $\implies x_B^{n-1} = f$ (from eq. (3)) $\implies flag_B^{n-1} = 0$ (from lemma 1) $\implies c_A^{n-1} = 1$ (from eq. (1)) $\implies x_A^n = e$ (from eq. (2)).

Similarly it can be proved for the case of $x_B^m = h$.

Theorem 4 (C4). $(x_A^n, x_B^n) \notin \{e, f\} \times \{e, f\}$ *for all $n > 0$.*

Proof. Let us assume that at some point of time, both processes are in the critical section. Let n be the first m for which $x_A^m \in \{e, f\} \wedge x_B^m \in \{e, f\}$, which means that $x_A^n \in \{e, f\} \wedge x_B^n \in \{e, f\}$ and $\forall i < n,\ x_A^i \notin \{e, f\} \vee x_B^i \notin \{e, f\}$.

1. Case 1: At $n - 1$, both processes are outside the critical sections.

$$x_A^{n-1} \notin \{e,f\} \wedge x_B^{n-1} \notin \{e,f\}$$

$$\begin{aligned}
x_A^{n-1} \notin \{e,f\} \wedge x_A^n \in \{e,f\} &\implies x_A^{n-1} = h \wedge c_A^{n-1} = 1 && \text{(From eq. (2))}\\
x_B^{n-1} \notin \{e,f\} \wedge x_B^n \in \{e,f\} &\implies x_B^{n-1} = h \wedge c_B^{n-1} = 1 && \text{(From eq. (3))}\\
x_A^{n-1} = x_B^{n-1} = h &\implies flag_A^{n-1} = flag_B^{n-1} = 1 && \text{(From lemma 1)}\\
&\implies c_A^{n-1} = 0 \vee c_B^{n-1} = 0 && \text{(From eq. (1))}\\
&\implies \text{Contradiction}
\end{aligned}$$

2. Case 2: At $n-1$, one of the processes (say P_B) is outside the critical section, i.e.,

$$x_A^{n-1} \in \{e,f\} \wedge x_B^{n-1} \notin \{e,f\}$$

$$\begin{aligned}
x_B^{n-1} \notin \{e,f\} \wedge x_B^n \in \{e,f\} &\implies x_B^{n-1} = h \wedge c_B^{n-1} = 1 && \text{(From eq. (3))}\\
&\implies flag_B^{n-1} = 1 && \text{(From lemma 1)}
\end{aligned}$$

   a. Subcase 1: $c_A^{n-1} = 1$
      If $x_A^{n-1} = f$, then $x_A^n = t$. Therefore, x_A^{n-1} cannot be f.

$$\begin{aligned}
x_A^{n-1} = e &\implies flag_A^{n-1} = 1 && \text{(From lemma 1)}\\
flag_A^{n-1} = flag_B^{n-1} = 1 &\implies c_A^{n-1} = 0 \vee c_B^{n-1} = 0 && \text{(From eq. (1))}\\
&\implies \text{Contradiction}
\end{aligned}$$

   b. Subcase 2: $c_A^{n-1} = 0$
      i. $x_A^{n-1} = f$

$$\begin{aligned}
&\implies flag_A^{n-1} = 0 && \text{(From lemma 1)}\\
&\implies c_A^{n-1} = 1 && \text{(From eq. (1))}\\
&\implies \text{Contradiction}
\end{aligned}$$

      ii. $x_A^{n-1} = e$

$$\begin{aligned}
&\implies flag_A^{n-1} = 1 && \text{(From lemma 1)}\\
&\text{Also, } turn = A && \text{(By careful analysis)}\\
flag_A^{n-1} = 1 \wedge flag_B^{n-1} = 1 \wedge turn = A &\implies c_A^{n-1} = 1 && \text{(From eq. (1))}\\
&\implies \text{Contradiction}
\end{aligned}$$

Thus, it is not possible that $x_A^n \in \{e,f\} \wedge x_B^n \in \{e,f\}$ for any n. Hence proved.

5 Related Work

There have been many works on the problem of mutual exclusion, the first one being by Dijkstra[7]. Among the few important ones that followed are Dekker's algorithm[8], Peterson's algorithm[12], Lamport's algorithm[11], etc. In this paper, we have chosen Peterson's algorithm for detailed analysis. Following are some of the ways in which the algorithm has been modelled in the literature.

There are many works that focus on improving the efficiency of the original algorithm. Block et al.[4] present a solution that improves the number of operations in the case of n processes competing for the critical section (the generalised version of the algorithm). A simple modification to the same is made by Alagarsamy[1] to bring a further minor improvement. As we are interested in the architecture aspect and not the efficiency, we do not go deeper into such works.

Reisig[13] uses Petri Nets to model the algorithm. Vaziri et al.[14] model the algorithm in Promela and do model checking using SPIN. Cicirelli et al.[6] model the algorithm using UPPAAL toolbox. Calculus of Communicating Systems (CCS) has been used by Dyseryn et al.[9] for modelling. However, none of them talk about modular composition of smaller components and equational proofs.

An approach that is closely related to ours is that of Attiogbe[3] in which he specifies several subsystems compose them to achieve mutual exclusion using Event B. His work differs from ours in that each subsystem is modelled as an abstract system, and not as a transition system, where there are 'events' in each abstract system that cooperate to achieve a task. Also, the ideas of feedback control as well as equational reasoning are not used. The same hold true for the event-based approach used by Ivanov et al.[10] in which several events are ordered using precedence relations and Peterson's algorithm is modelled and proved.

Another related approach is used by Arnold[2] where he uses transition systems to model processes and has the similar notion transitions representing the state change of processes. But the interaction between processes, or how multiple processes are composed to each other to form the bigger system is not explained. Feedback control is not used.

6 Conclusion

The main objective of this paper is to show how we can systematically apply the idea of transition systems and composition in the form of feedback loop to solve the mutual exclusion problem, for which we chose Peterson's algorithm for demonstration. The result is a very modular solution with interconnected components, whose dynamics can be explained by simple equational reasoning. Therefore, the transition systems approach is effective, and is a promising direction to explore. We look forward to applying the idea to more such examples in the future.

References

[1] Alagarsamy K (2005) A mutual exclusion algorithm with optimally bounded bypasses. Information Processing Letters 96(1):36–40

[2] Arnold A (1989) Mec: a system for constructing and analysing transition systems. In: International Conference on Computer Aided Verification, Springer, pp 117–132

[3] Attiogbé JC (2005) A stepwise development of the petersons mutual exclusion algorithm using b abstract systems. In: International Conference of B and Z Users, Springer, pp 124–141

[4] Block K, Woo TK (1990) A more efficient generalization of peterson's mutual exclusion algorithm. Information Processing Letters 35(5):219–222

[5] Chilton C, Jonsson B, Kwiatkowska MZ (2012) Assume-guarantee reasoning for safe component behaviours. In: FACS, Springer, vol 12, pp 92–109

[6] Cicirelli F, Nigro L (2016) Modelling and verification of mutual exclusion algorithms. In: Distributed Simulation and Real Time Applications (DS-RT), 2016 IEEE/ACM 20th International Symposium on, IEEE, pp 136–144

[7] Dijkstra EW (1965) Solution of a problem in concurrent programming control. In: Pioneers and Their Contributions to Software Engineering, Springer, pp 289–294

[8] Dijkstra EW (1968) Cooperating sequential processes. In: The origin of concurrent programming, Springer, pp 65–138

[9] Dyseryn V, van Glabbeek R, Höfner P (2017) Analysing mutual exclusion using process algebra with signals. arXiv preprint arXiv:170900826

[10] Ivanov I, Nikitchenko M, Abraham U (2015) Event-based proof of the mutual exclusion property of petersons algorithm. Formalized Mathematics 23(4):325–331

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

[12] Peterson GL (1981) Myths about the mutual exclusion problem. Information Processing Letters 12(3):115–116

[13] Reisig W (1995) Correctness proofs of distributed algorithms. Theory and Practice in Distributed Systems pp 164–177

[14] Vaziri M, Holzmanny G (1998) Automatic generation of invariants in spin. Proc of the Int SPIN Work(SPIN98)