# Model Checking classical mutual exclusion algorithms

Ivan Ivchenko

01/08/2024

## 0.1 Brief introduction

During this project we will take some of the mutual exclusion algorithms and perform a model-checking for them. The algorithms that were selected:

1. Dekker's algorithm
2. Peterson's algorithm

The tool that we are going to use is UPPAAL. The project and all of the files will also be published on my GitHub account (link).

## 0.2 Dekker's Algorithm

Dekker's algorithm is the first known correct solution to the mutual exclusion problem in concurrent programming where processes only communicate via shared memory.

If two processes try to enter a critical section at the same time, the algorithm will only allow one of them to do so, based on whose turn it is at that moment. If one process has already entered the critical section, the other will wait for the first to leave. This is implemented by using two flags (indicators of "intent" to enter the critical section) and a turn variable (indicating which process has queued).

We can describe this using pseudocode:

**Algorithm 1:** Initialization

```
1 flag[0] := false
2 flag[1] := false
3 turn := 0 // or 1
```

**Algorithm 2:** Algorithm for process A

```
1  flag[0] := true
2  while flag[1] = true do
3  |  if turn = 1 then
4  |  |  flag[0] := false
5  |  |  while turn = 1 do
6  |  |  |  busy-wait
7  |  |  end
8  |  |  flag[0] := true
9  |  end
10 end
11 // critical section
12 ...
13 turn := 1
14 flag[0] := false
15 // end of critical section
16 ...
```

**Algorithm 3:** Algorithm for process B

```
1  flag[1] := true
2  while flag[0] = true do
3  |  if turn = 0 then
4  |  |  flag[1] := false
5  |  |  while turn = 0 do
6  |  |  |  busy-wait
7  |  |  end
8  |  |  flag[1] := true
9  |  end
10 end
11 // critical section
12 ...
13 turn := 0
14 flag[1] := false
15 // end of critical section
16 ...
```

Dekker's algorithm ensures mutual exclusion and prevents deadlock. If process A remains in the "while flag[1]" loop indefinitely, eventually process B will enter its critical section and set turn = 0. A will then exit its inner "while turn = 1" loop, set flag[0] = true, and wait for flag[1] to become false. When B tries to re-enter the critical section, it will set flag[1] = false and loop on "while turn = 0." This allows A to proceed into its critical section. Removing the "turn = 0" check in A's loop would introduce the possibility of starvation, so all steps are crucial.

Because we are using UPPAAL GUI, in the next images I will present the Dekker's algorithm loaded into the tool and how i defined it.

Before we begin we need to declare global variables and our system specifications.

```
1 bool flag[2] =
     {false, false};
2 int turn = 1;
```

Figure 1: Global variables declarations

```
1 P0 = proc(0);
2 P1 = proc(1);
3 system P0, P1;
```

Figure 2: System declarations

```
1 const int pid
```

Figure 3: Process parameter

Now moving on to the process definition (Figure 4). For this, our model with states (locations) that correspond to actions in the pseudocode, the guard statements (what should hold in order to move from one state to another) are marked green and the update statements (what values will be changed after the step) are marked blue.

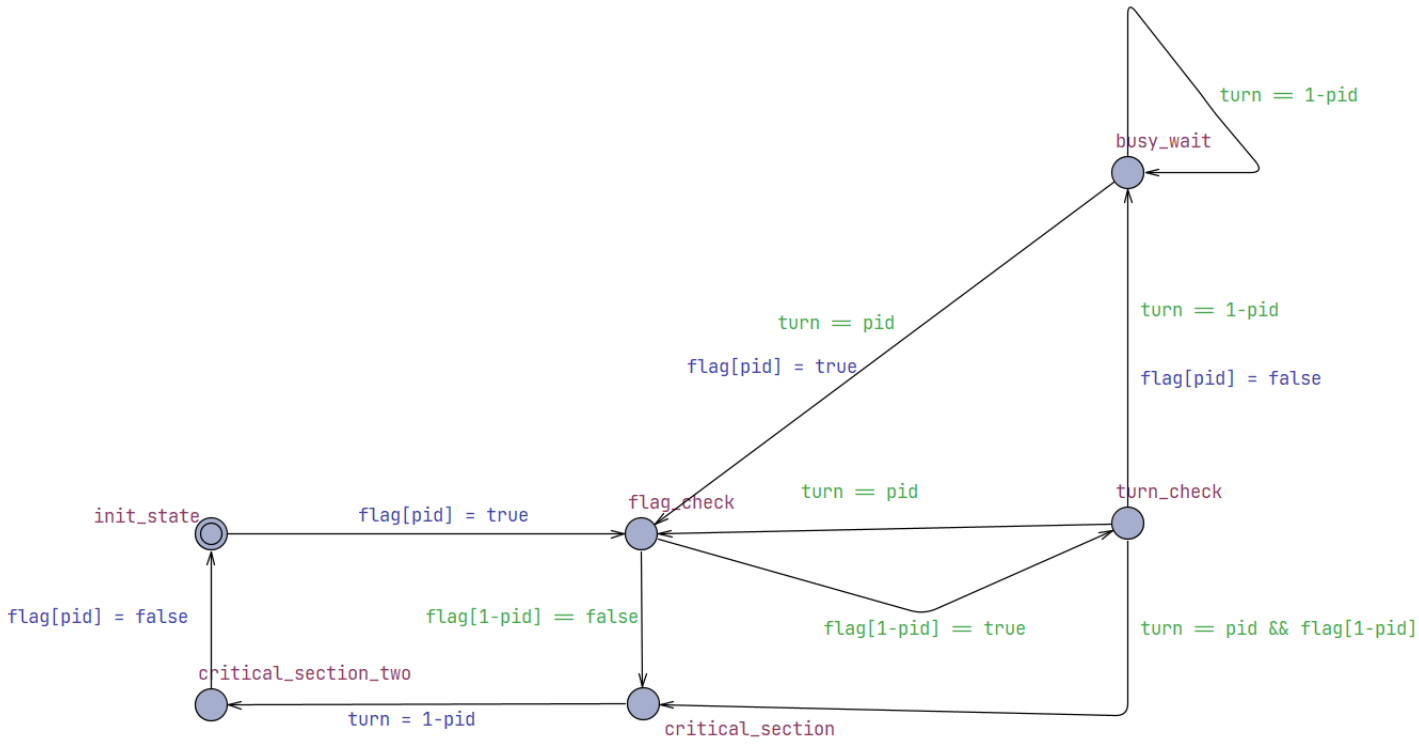


Figure 4: Definition of our processes

For the properties to be verified we specify:

1. $A[]$ not $(P0.\text{critical\_section} \wedge P1.\text{critical\_section}) \wedge$
   not $(P0.\text{critical\_section\_two} \wedge P1.\text{critical\_section}) \wedge$
   not $(P1.\text{critical\_section\_two} \wedge P0.\text{critical\_section}) \wedge$
   not $(P1.\text{critical\_section\_two} \wedge P0.\text{critical\_section\_two})$ - with UPPAAL specification we use A[] symbol to represent "on all paths at all points in time" or we can rephrase it to "always", and because with our specification of the critical section that consists of 2 states (locations), we need to check for all possible combination of states that will trigger mutual exclusion

2. $A[]$ not deadlock - with UPPAAL specification we use A[] symbol to represent "on all paths at all points in time" or we can rephrase it to "always", this ensures that our model is defined and initialized correctly

3. $E[]$ $(P0.busy\_wait \vee P1.busy\_wait)$ - the symbol $E[]$ in the UPPAAL specification is used to denote "there exists a path on which this statement is executed indefinitely" or "in some path always". This query checks that there exists a path on which at least one process is stuck in the busy-wait forever (livelock).

4. $E <>$ not$(P0.busy_wait \wedge P1.busy_wait)$ - the symbol $E <>$ in the UPPAAL specification means "there exists a path on which this statement is executed". In this case, the query checks if there is a path on which not both processes are idle at the same time. In other words, this query checks that there is no path on which both processes are constantly in an busy-wait state at the same time.

After this, all we have to do is to run a check to see if our properties are being satisfied.

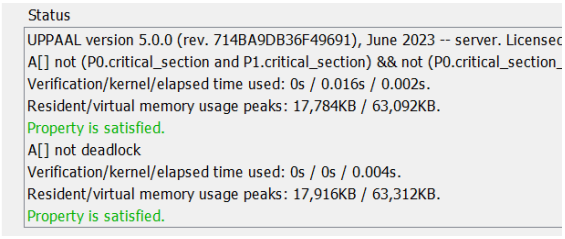


Figure 5: Results of our check 1

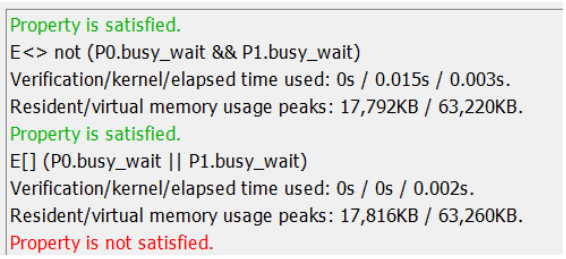


Figure 6: Results of our check 2

Explanation:

The query $A[]$ not $(P0.\text{critical\_section} \wedge P1.\text{critical\_section}) \wedge$ not $(P0.\text{critical_section_two} \wedge P1.\text{critical_section}) \wedge$not $(P1.\text{critical_section_two} \wedge P0.\text{critical_section}) \wedge$not $(P1.\text{critical\_section\_two} \wedge P0.\text{critical\_section\_two})$ holding true indicates that **it is always guaranteed** that **at no point in time** will both processes $P0$ and $P1$ be in their critical sections simultaneously. This confirms that the system maintains mutual exclusion, ensuring that **two processes cannot enter their critical sections at the same time**.

The query $A[]$ not deadlock holding true indicates that **the system is free from deadlock**. This means that in every possible execution, the system will never reach a state where all processes are indefinitely waiting and cannot proceed.

The query $E[]$ $(P0.busy\_wait \vee P1.busy\_wait)$ failing indicates that there are some states where neither P0 nor P1 is in the busy-wait state simultaneously. This suggests that the system avoids livelock, as it demonstrates that the processes **are not indefinitely stuck in busy-wait conditions without making progress**.

The query $E <>$ not$(P0.busy_wait \wedge P1.busy_wait)$ holding true indicating that there exists at least one state in which it is not true that both processes P0 and P1 are in the busy-wait state simultaneously. This result confirms that the system **avoids scenarios where both processes are perpetually stuck in busy-wait at the same time**.

## 0.3 Peterson's Algorithm (for 2 processes)

Peterson's algorithm (or Peterson's solution) is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.

The algorithm uses two variables: flag and turn. A flag[n] value of true indicates that the process n wants to enter the critical section. Entrance to the critical section is granted for process A if B does not want to enter its critical section or if B has given priority to A by setting turn to 0.

Using the pseudocode , we can present Peterson's algorithm like this:

**Algorithm 4:** Initialization

```
1 bool flag[2] = false, false;
2 int turn;
```

**Algorithm 5:** Algorithm for process A

```
1 flag[0] = true;
2 turn = 1;
3 while flag[1] == true and turn==1
    do
4 |  busy-wait
5 end
6 //critical section
7 .....
8 //end of critical section
9 flag[0] = false;
```

**Algorithm 6:** Algorithm for process B

```
1 flag[1] = true;
2 turn = 0;
3 while flag[0] == true and turn==0
    do
4 |  busy-wait
5 end
6 //critical section
7 .....
8 //end of critical section
9 flag[1] = false;
```

A and B can never be in the critical section at the same time. If A is in its critical section, then flag[0] is true. In addition, either flag[1] is false (meaning that B has left its critical section), or turn is 0 (meaning that B is just now trying to enter the critical section, but graciously waiting). So if both processes are in their critical sections, then we conclude that the state must satisfy flag[0] and flag[1] and turn = 0 and turn = 1. No state can satisfy both turn = 0 and turn = 1, so there can be no state where both processes are in their critical sections.

Because we are using UPPAAL GUI, in the next images I will present the Peterson's algorithm loaded into the tool and how i defined it.

Now moving on to the process definition. For this, our model with states (locations) that correspond to actions in the pseudocode. For better understanding:

```
bool flag[2] =
    {false, false};

int[0,1] turn;
```

Figure 7: Global variables declarations

```
P0 = proc(0);
P1 = proc(1);
system P0, P1;
```

Figure 8: System declarations

```
const int pid
```

Figure 9: Process parameter

Now moving on to the process definition. For this I've simplified our model for a better understanding of the concept of algorithm, the guard statements (what should hold in order to move from one state to another) are marked green and the update statements (what values will be changed after the step) are marked blue (Figure 9).

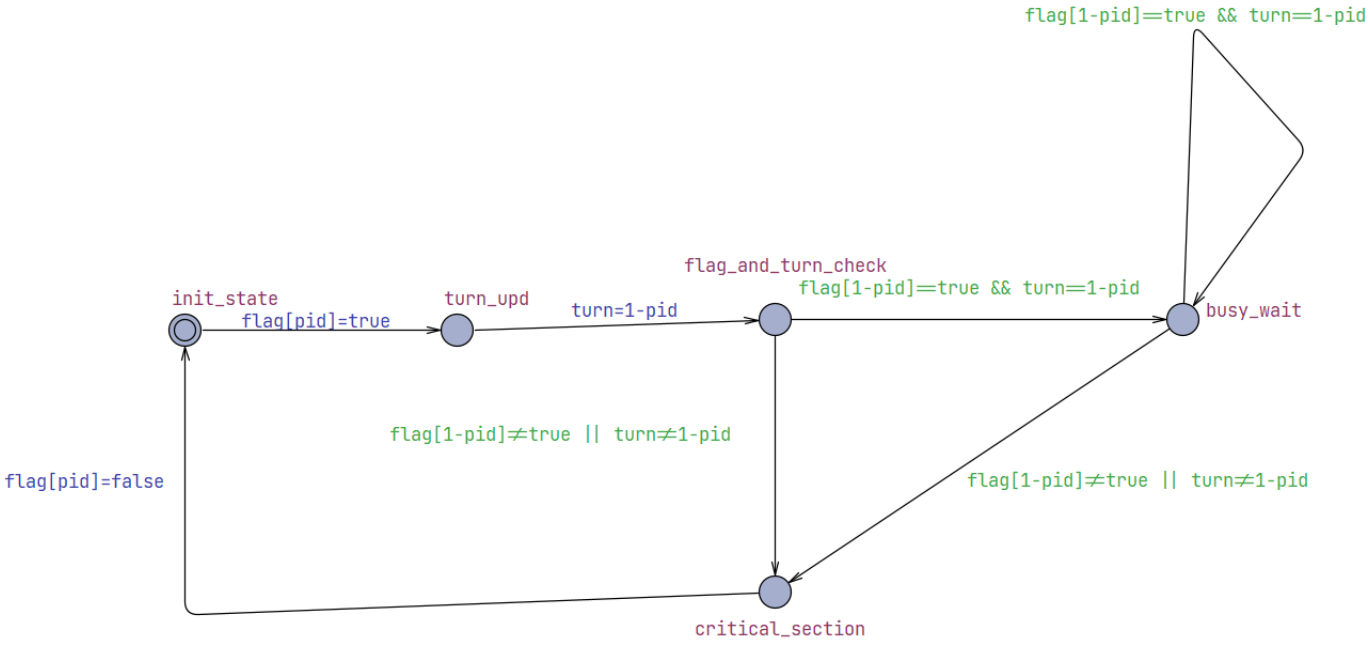


Figure 10: Definition of our processes

For the properties to be verified we specify:

1. $A[]$ not $(P0.\text{critical\_section} \wedge P1.\text{critical\_section})$ - with UPPAAL specification we use A[] symbol to represent "on all paths at all points in time" or we can rephrase it to "always", with this we check for the mutual exclusion.

2. $A[]$ not deadlock - with UPPAAL specification we use A[] symbol to represent "on all paths at all points in time" or we can rephrase it to "always", this ensures that our model is defined and initialized correctly

3. $E[]$ $(P0.busy\_wait \vee P1.busy\_wait)$ - the symbol $E[]$ in the UPPAAL specification is used to

denote "there exists a path on which this statement is executed indefinitely" or "in some path always". This query checks that there exists a path on which at least one process is stuck in the busy-wait forever (livelock).

4. $E <>$ not$(P0.busy_wait \wedge P1.busy_wait)$ - the symbol $E <>$ in the UPPAAL specification means "there exists a path on which this statement is executed". In this case, the query checks if there is a path on which not both processes are idle at the same time. In other words, this query checks that there is no path on which both processes are constantly in an busy-wait state at the same time.

After this, all we have to do is to run a check to see if our properties are being satisfied.

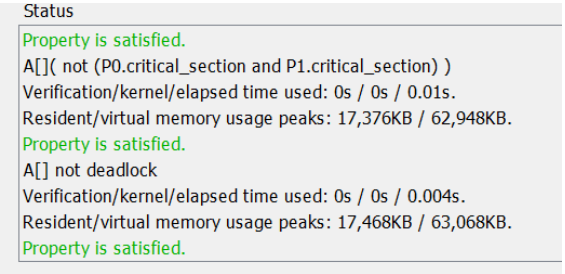


Figure 11: Results of our check 1

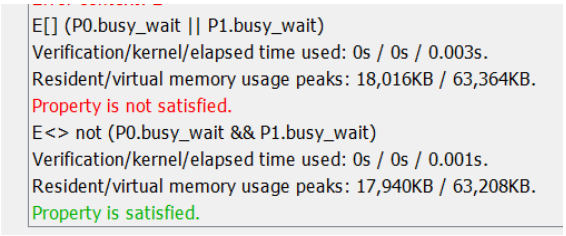


Figure 12: Results of our check 2

Explanation:

The query $A[]$ not $(P0.\text{critical\_section} \wedge P1.\text{critical\_section})$ holding true indicates that **it is always guaranteed** that **at no point in time** will both processes $P0$ and $P1$ be in their critical sections simultaneously. This confirms that the system maintains mutual exclusion, ensuring that **two processes cannot enter their critical sections at the same time**.

The query $A[]$ not deadlock holding true indicates that **the system is free from deadlock**. This means that in every possible execution, the system will never reach a state where all processes are indefinitely waiting and cannot proceed.

The query $E[]$ $(P0.busy\_wait \vee P1.busy\_wait)$ failing indicates that there are some states where neither P0 nor P1 is in the busy-wait state simultaneously. This suggests that the system avoids livelock, as it demonstrates that the processes **are not indefinitely stuck in busy-wait conditions without making progress**.

The query $E <>$ not$(P0.busy_wait \wedge P1.busy_wait)$ holding true indicating that there exists at least one state in which it is not true that both processes P0 and P1 are in the busy-wait state simultaneously. This result confirms that the system **avoids scenarios where both processes are perpetually stuck in busy-wait at the same time**.

## 0.4 Conclusion

In this project, we successfully modeled and verified Dekker's and Peterson's mutual exclusion algorithms using UPPAAL. Through the creation of process templates, the definition of shared variables, and the formulation of transitions and guard conditions, we were able to represent the core logic of these algorithms accurately.

We verified critical properties such as mutual exclusion and absence of deadlock by utilizing UPPAAL's query capabilities. The mutual exclusion property ensured that no two processes could be in the critical section simultaneously, while the absence of deadlock guaranteed that processes would not get stuck indefinitely waiting to enter the critical section. Through this project, we learned how to use the tool for model-checking and also familiarized ourselves with the basic mutual exclusion algorithms.